The Report Committee for Sumant Dalmiya
Certifies that this is the approved version of the following report:

A Comparative Study of Adders

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor: _____
Earl E. Swartzlander Jr.

_____
Nur A. Touba

A Comparative Study of Adders

by

Sumant Dalmiya, B.E.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2015

Abstract

A Comparative Study of Adders

Sumant Dalmiya, MSE

The University of Texas at Austin, 2015

Supervisor: Earl E. Swartzlander Jr.

This report compares the area, delay, complexity (in terms of gate count) and power of 16, 32 and 64 bit versions of different types of serial and parallel adders. Ripple carry adder, Carry look-ahead adder, carry select adder and parallel prefix adders like Brent-Kung, Kogge-Stone, Han-Carlson and Ladner-Fischer were studied. For the parallel adders schematics were designed in Cadence Virtuoso Schematic Editor using 2 input NAND, NOR and INVERTER gate as the standard cells in 45nm technology. The other adders were implemented using structural Verilog and synthesized using Design Vision (by Synopsys). Auto Place and Route was performed using Cadence Encounter to get the layout of the adders and then Parasitic Extraction was performed to get the actual routing delay. Post-PNR netlist was used to compare the area, delay and power of the various adders. Area-Delay product was used as a figure of merit to compare the adders.

Table of Contents

List of Tables

List of Figures

1. Introduction

The simplest form of adder is the ripple-carry adder. An n-bit ripple carry adder consists of n one-bit full adders connected in succession. The carry "ripples" from the least significant bit to the most significant bit and hence the name. Since, the carry in at any stage depends on the carry out from the previous stage, the delay of ripple-carry adders is O (N) and increases linearly as the size of the operands is increased. This becomes inadequate as the size of the operands increases to 64 bits and 128 bits. Thus it is essential to look for other alternates.

Carry select adders [1] consist of two ripple carry adders and a multiplexer. Addition of two n-bit numbers is done with two adders: one time with the assumption that the carry in is zero, and the other assuming that the carry is one. After the two results are calculated, the correct sum and the correct carry is selected with the multiplexer once the correct carry of previous stage is known. The number of bits in each carry select block can be uniform, or variable. In the uniform case, the optimal delay occurs for a block size of $\sqrt{N}$, where N is the size of the operands. The carry select adders are simple, but rather fast compared to the ripple carry adders having a delay of O ($\sqrt{N}$).

The higher delay in the case of ripple carry adders is due to the carry chain. In carry look-ahead adders, the carry signals are calculated in advance, based on the input signals [2], [3]. For any bit position i, a carry will be generated if the corresponding input bits are '1' or if the carry-in to that bit was a '1' and at least one of the input bits are '1'. From this, a recurrence relation is derived that expresses carry in to any bit position in terms of the relevant addend and augend digits and some lower-adder carry. This can result in considerable gain in speed.

For wide adders, the delay of the carry-lookahead adders becomes dominated by the delay of passing the carry through the lookahead stages. This delay can be reduced by looking ahead across the look-ahead blocks [4]. In general, a multi-level tree of lookahead structures can be generated to achieve delay that grows with (Log N). All parallel prefix adders are parallel "carry look-

ahead" adders suitable for implementation in VLSI architecture. Using parallel-prefix adders, addition can be performed in time O(log n) using area O(n log n). Various types of parallel-prefix adders have been developed to minimize the delay of the critical path by performing the execution of operations in parallel. This is done by segmentation of the operation into smaller pieces that are done in parallel.

Kogge-Stone [5] uses a technique called recursive doubling in an algorithm for solving a large class of recurrence problems on parallel computers. Recursive doubling involves the splitting of the computation of a function into two equally complex sub functions whose evaluation can be performed simultaneously. The Kogge-Stone adder has $\log_2 N$ stages and a fanout of 2 at each stage. This comes at the cost of many long wires that must be routed between stages.

Ladner-Fischer [6] provides a general method for deriving efficient parallel solutions to the fixed-length version of any problem solved by a finite state transducer. The Ladner-Fischer adder computes prefixes for the odd numbered bits and uses one more stage to ripple into the even positions. Cells at high-fanout nodes must still be sized or ganged appropriately to achieve good speed.

Brent-Kung [7] is a simple parallel adder with regular design. It takes into account the problem of connecting the gates in an economical and regular way to minimize chip area and design costs along with delay and complexity. It uses the same idea as Ladner-Fischer but is not directly applicable because they ignored fan-out restriction. Brent-Kung assumes existence of gates that compute a logical function of two inputs in constant time and an output signal can be divided into two signals in constant time.

Han-Carlson [8] is a new graph representation for prefix computation that leads to the design of a fast, area-efficient binary adder. The new graph is a combination of Brent-Kung and Kogge-Stone for prefix computation, and its area is close to known lower bounds on the VLSI area of parallel prefix graphs. Han-Carlson adders perform Kogge-Stone on the odd numbered bits, and then use

one more stage to ripple into the even positions. They claim it to be the fastest possible area-efficient VLSI adder.

In this report the delay and complexity of 16 bit, 32 bit and 64 versions of the following types of adders were compared:

- Ripple Carry
- Carry Look-ahead
- Carry Select
- Brent-Kung
- Kogge-Stone
- Han-Carlson
- Lardner-Fischer

Subtraction and overflow detection have also been implemented in the adders by using 2's complement arithmetic and the functional correctness of the adders was verified using Verilog-XL and Modelsim.

2. Methodology

A structural Verilog code was written to implement the 16 bit, 32 bit and 64 bit ripple-carry, carry lookahead and the carry select adders whereas for parallel prefix adders schematics were made using Cadence Virtuoso. Since the same technology file and the standard cells were used for both the techniques, there wouldn't be any major source of discrepancy. All the adders had the **subtraction** functionality as well. For this, an input add_sub was used to indicate which operation was to be performed. For addition, add_sub = 0 and for subtraction add_sub = 1. A 2:1 multiplexer was used to determine the second input to the adder (B or ~B) based on add_sub. The signal add_sub was also connected to C_{in} since to get –B, 1 is to be added to the 1's complement of B.

Overflow detection was implemented by using XOR gates. Overflow occurs when addition of two positive numbers results in a negative output or the addition of 2 negative numbers result in a positive output. The carry in to the MSB of the bits was XORed with the carry out from the last bit position to get the overflow bit.

Once the structural Verilog was verified for correct functionality, it was synthesized using Design Vision by Synopsys. The gate level netlist obtained after synthesis was used for Auto Place and Route (APR). For parallel prefix adders, since the design entry was done using schematic, the netlist obtained from the schematic could be directly used for APR. Cadence Encounter tool was used to Auto Place and Route (APR) and get the layout for these adders. Multiple iterations were performed so that the layout obtained had minimum area as well as delay.

2.1. Gate Level Design

Ripple Carry Adder

Ripple carry adders are made of full-adders with the carry bit propagating from the LSB to the MSB. A simple block diagram of a 4-bit ripple carry adder is

shown below. A[3:0] and B[3:0] are the two operands and C0 is the carry-in. c1, c2, and c3 are the carry bits that are propagated from LSB towards MSB and c4 is the carry-out of the 4 bit adder. S[3:0] is the sum of A and B.

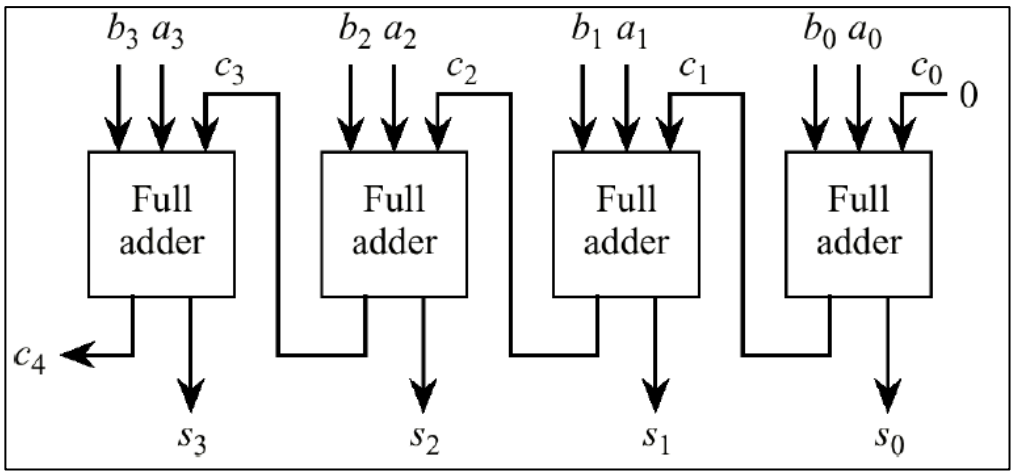

Figure 1: Block Diagram of a 4-bit Ripple Carry Adder

A full adder module was created and instantiated 4 times to make a 4 bit ripple carry adder. Four 4-bit ripple carry adders were connected together to make a 16 bit adder. Two 16 bit adders were used to make a 32-bit adder and four 16 bit adders were used to make a 64-bit adder. The structural Verilog code was then tested to ensure that the functionality was correct.

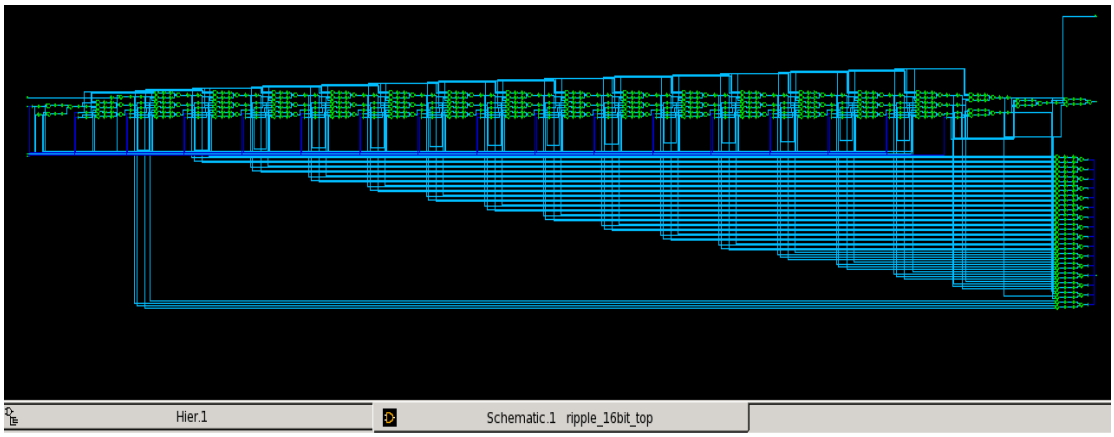

Figure 2: Gate level representation of a 16-bit Ripple Carry Adder

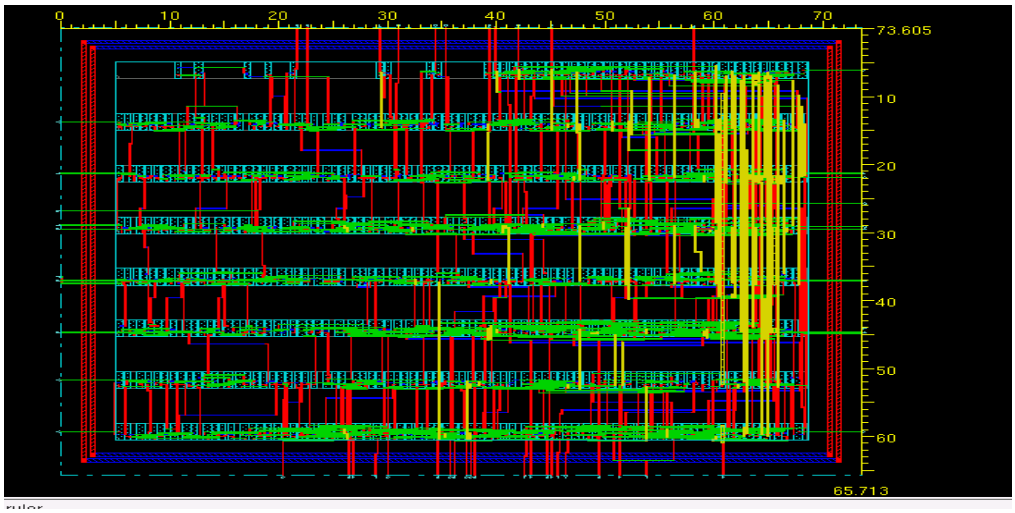

Figure 3: Layout of a 16-bit Ripple Carry Adder

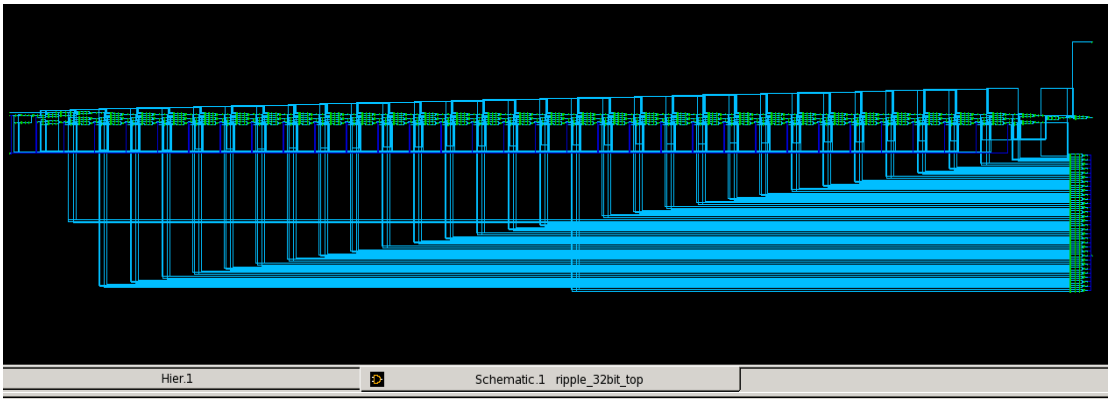

Figure 4: Gate level representation of a 32-bit Ripple Carry Adder

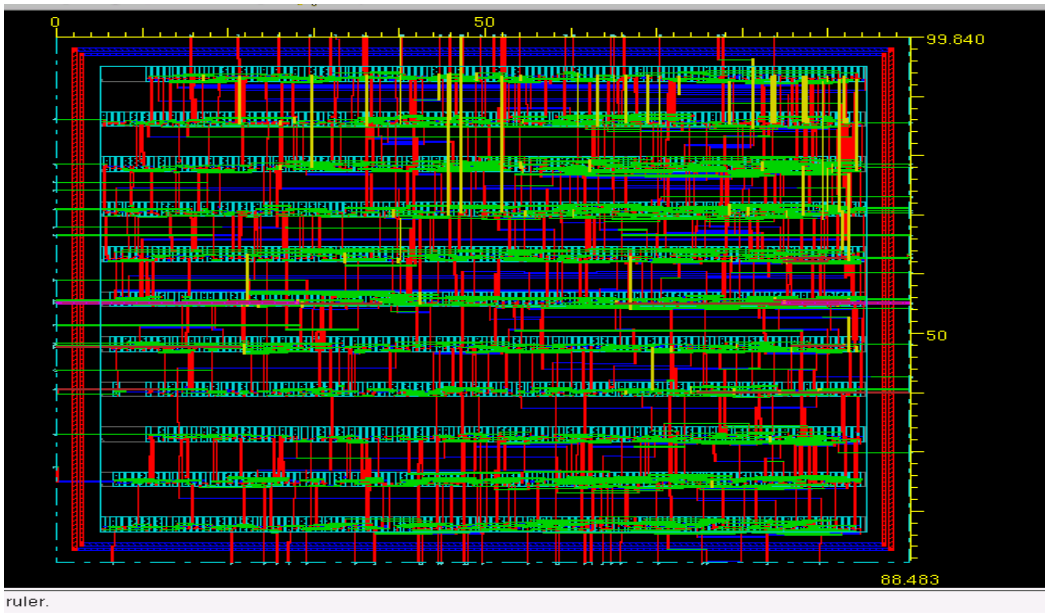

Figure 5: Layout of a 32-bit Ripple Carry Adder

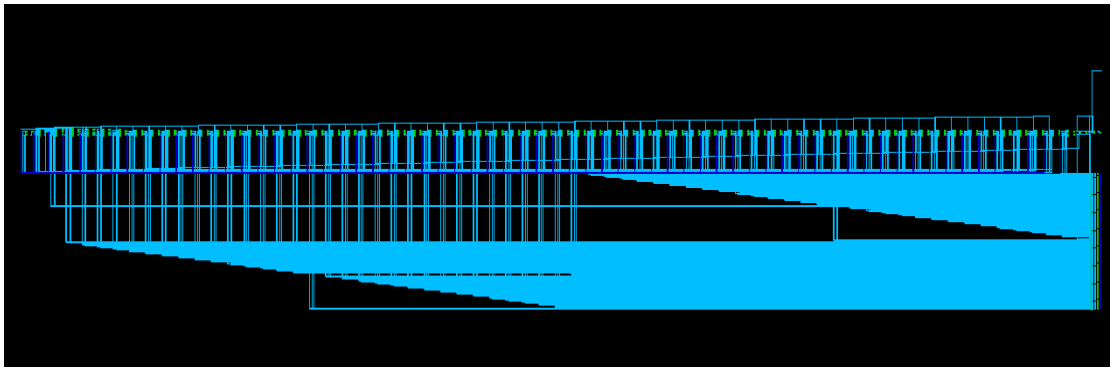

Figure 6: Gate level representation of a 64-bit Ripple Carry Adder

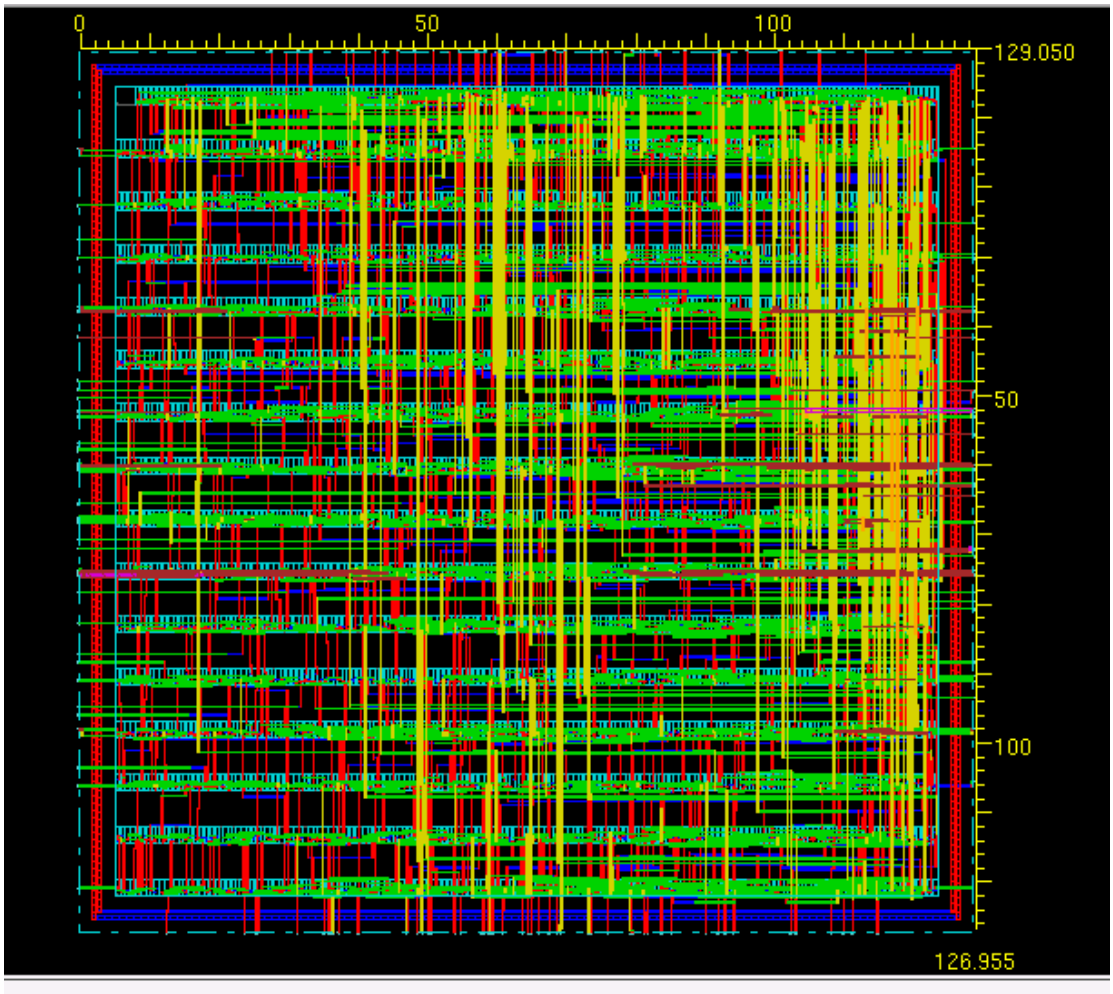

Figure 7: Layout of a 64-bit Ripple Carry Adder

Carry Select Adder

Carry select adder is made up of blocks of ripple carry adders. Two Ripple carry adders are used, one with carry in of 0 and 1 with carry in of 1, to generate the sum and the carry out bits. Depending on the carry out from the previous stage, the correct sum and carry out bits are chosen using a multiplexer. The basic structure of a 16 bit carry select adder is shown below.

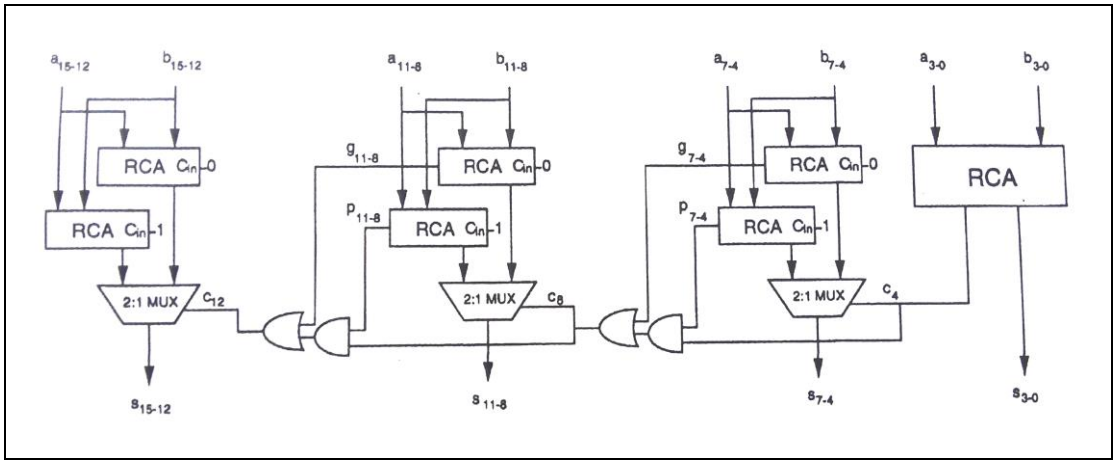

Figure 8: Block Diagram of a 16-bit Carry Select Adder

For 16 bit carry select adder, 4 bit ripple carry adders were used as basic blocks. If uniform sized ripple carry adders are used, the delay for carry select adder is minimum is the size of each ripple carry adder is $\sqrt{N}$. Hence, a 4 bit ripple carry adder module was created and multiple copies of the same were used along with multiplexers to get a 16 bit carry select adder. Similarly, 8 bit ripple carry adder blocks were used for 64 bit carry select adders. For 32 bit carry select adder, various experiments were done both with uniform size and variable size ripple carry blocks. The following different block sizes were used (from MSB to LSB):

1. Uniform sized 4 bit RCA Blocks
2. Uniform sized 8 bit RCA Blocks
3. (MSB) 2,6,6,6,6,6 (LSB) bit RCA blocks
4. (MSB) 6,7,6,5,4,4 (LSB) bit RCA blocks

5. (MSB)2,3,4,5,6,5,4,3 (LSB) bit RCA blocks

The carry select adder made with variable sized blocks of RCA (msb 6,7,6,5,4,4 lsb) was found to have the minimum delay. This was expected because it has only 6 stages of ripple carry adders and the RCA block in the critical path has only 4 bits. By the time the larger blocks of size 6, 7 compute the sum and carry, the correct carry would be propagated from the previous stages and hence the delay is minimized.

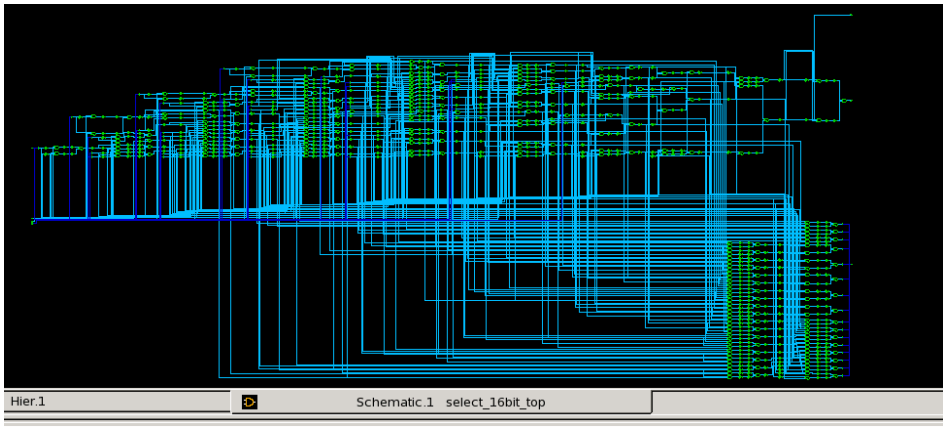

Figure 9: Gate level representation of a 16-bit Carry Select Adder

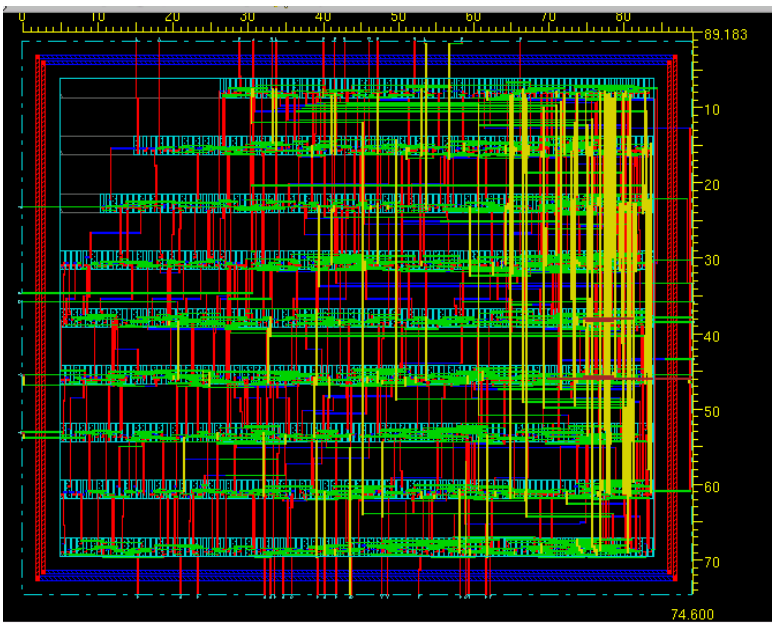

Figure 10: Layout of a 16-bit Carry Select Adder

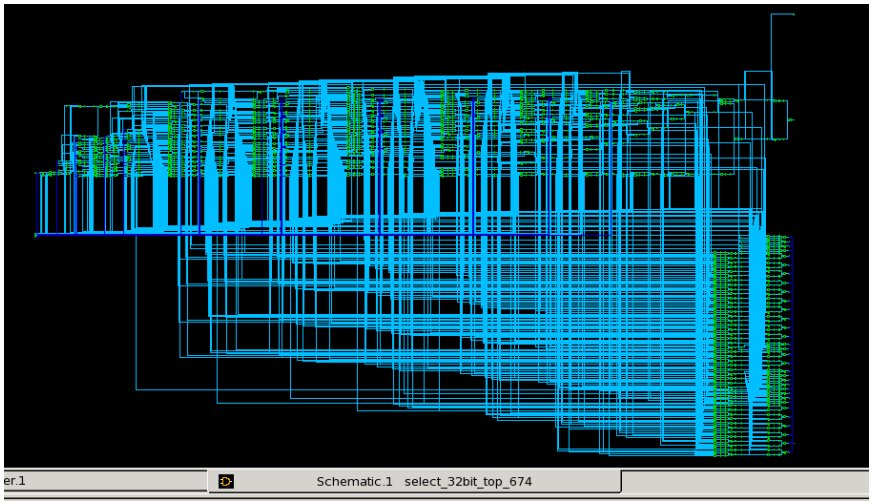

Figure 11: Gate level representation of a 32-bit Carry Select Adder

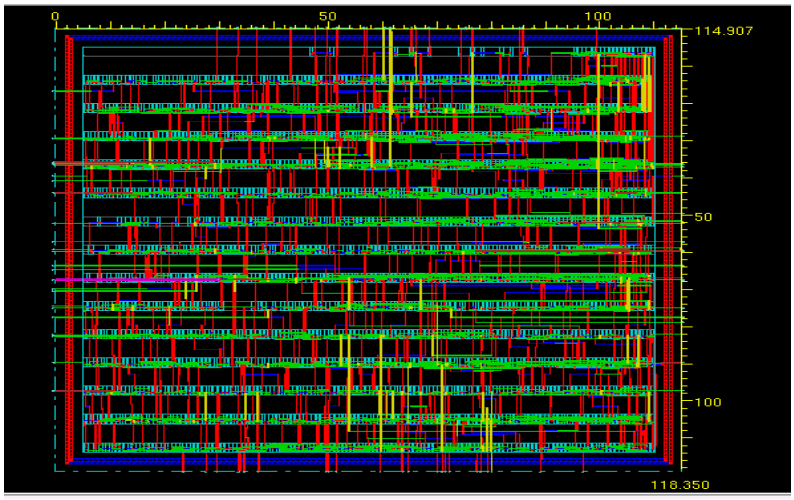

Figure 12: Layout of a 32-bit Carry Select Adder

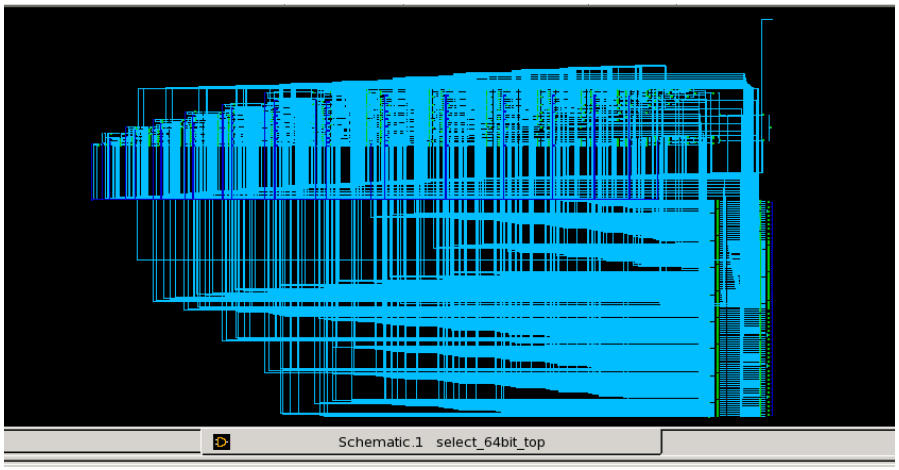

Figure 13: Gate level representation of a 64-bit Carry Select Adder

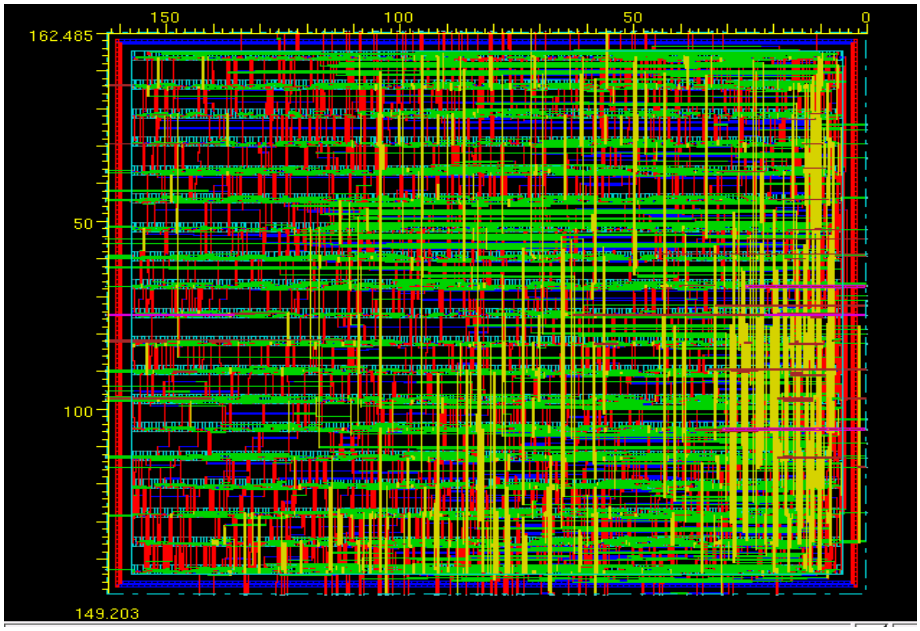

Figure 14: Layout of a 64-bit Carry Select Adder

Carry Lookahead Adder

The figure below shows the structure of a 4-bit carry lookahead adder. It consists of 4 modified full adders and a carry lookahead logic block. The carry lookahead block generates the carry signals c1, c2 and c3 for the 4 bit adder and also the group generate and propagate signal.

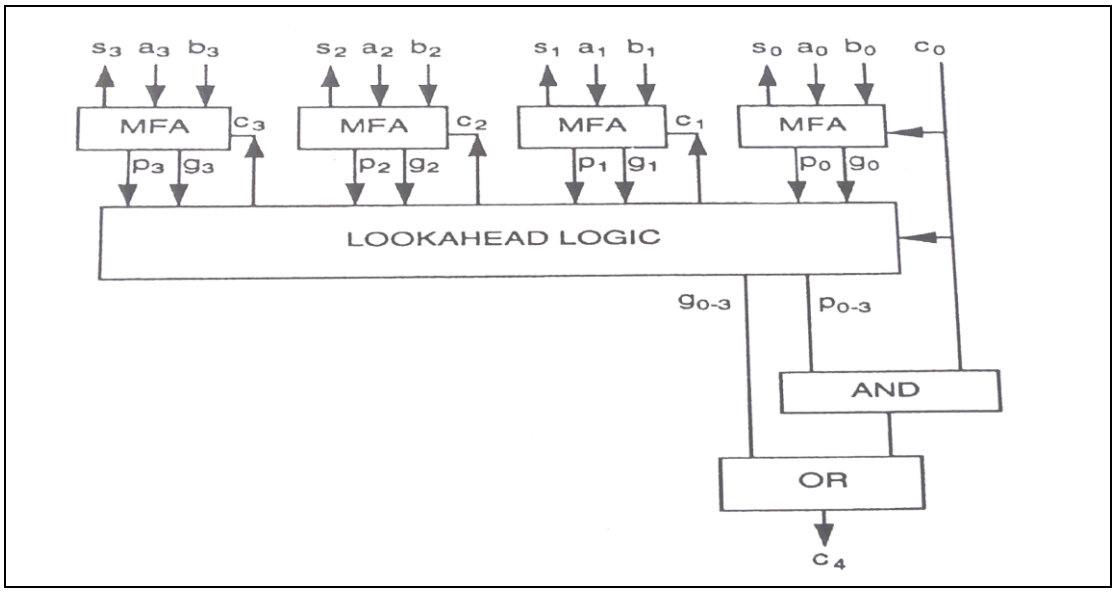

Figure 15: Block Diagram of a 4-bit Carry Lookahead Adder

The carry lookahead block generates the carry signals and the group

propagate and generate signals as given by the following equations:

$$g_k = a_k b_k \text{ and } p_k = a_k + b_k$$

$$c_{k+1} = g_k + p_k c_k$$

$$\begin{aligned} c_{k+2} &= g_{k+1} + p_{k+1} c_{k+1} \\ &= g_{k+1} + p_{k+1}(g_k + p_k c_k) \\ &= g_{k+1} + p_{k+1} g_k + p_{k+1} p_k c_k \end{aligned}$$

$$\begin{aligned} c_{k+3} &= g_{k+2} + p_{k+2} c_{k+2} \\ &= g_{k+2} + p_{k+2}(g_{k+1} + p_{k+1} g_k + p_{k+1} p_k c_k) \\ &= g_{k+2} + p_{k+2} g_{k+1} + p_{k+2} p_{k+1} g_k + p_{k+2} p_{k+1} p_k c_k \end{aligned}$$

$$g_{k+3:k} = g_{k+3} + p_{k+3} g_{k+2} + p_{k+3} p_{k+2} g_{k+1} + p_{k+3} p_{k+2} p_{k+1} g_k$$

$$p_{k+3:k} = p_{k+3} p_{k+2} p_{k+1} p_k$$

Figure 16: Carry Lookahead Adder Equations

Carry out from the 4 bit adder is given by the following equation:

$C_{out}=C_{in}.P_{0\text{-}3} + G_{0\text{-}3}$

The group generate and propagate signals can be used to cascade the 4 bit adders in order to get a 16 bit adder. This would require having 4 4-bit CLA adders and an additional carry lookahead block as shown in the figure below.

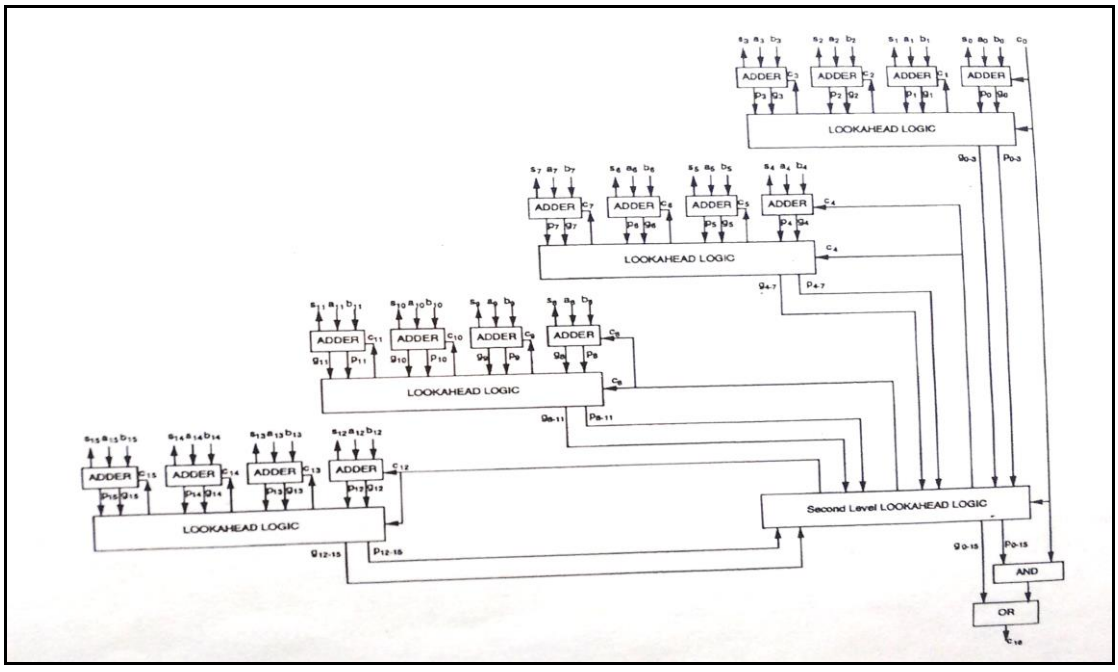

Figure 17: Block Diagram of a 16-bit Carry Lookahead Adder

32 bit and 64 bit carry look-ahead adder can be similarly made from 16 bit carry lookahead adders.

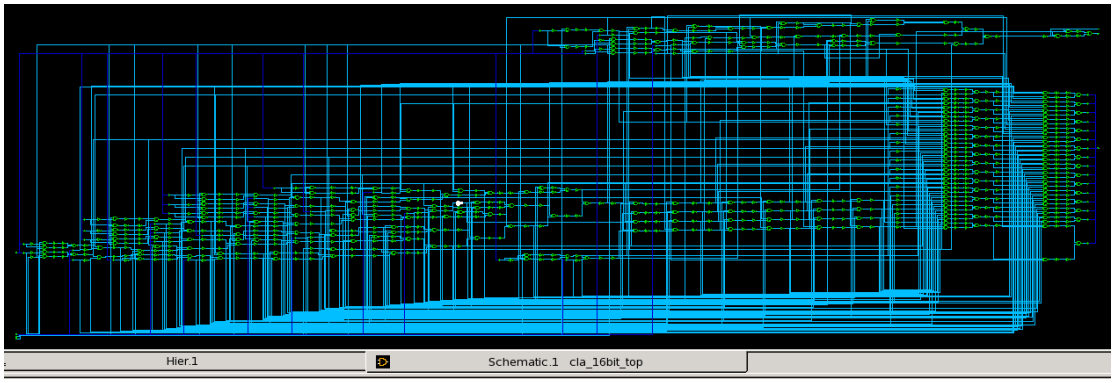

Figure 18: Gate level representation of a 16-bit Carry Lookahead Adder

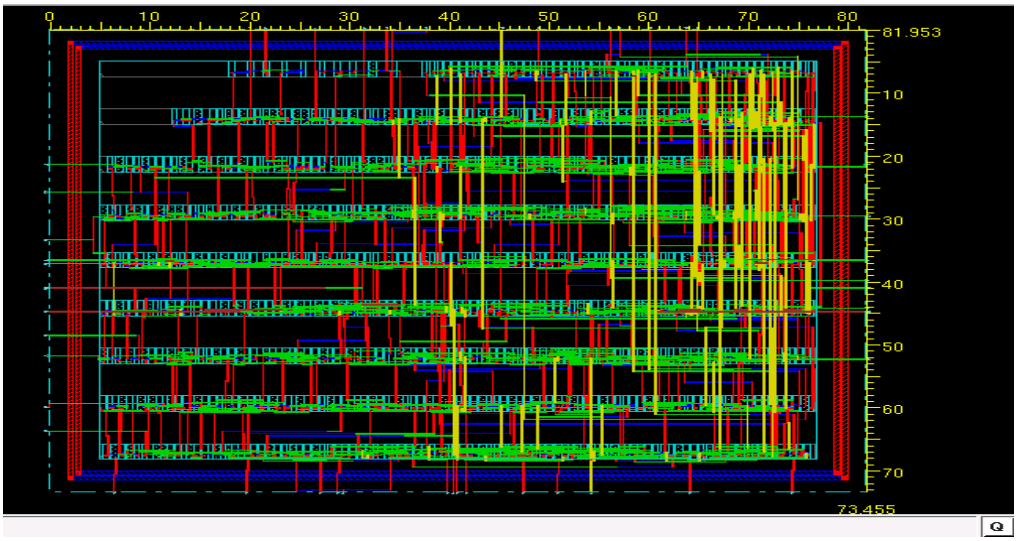

Figure 19: Layout of a 16-bit Carry Lookahead Adder

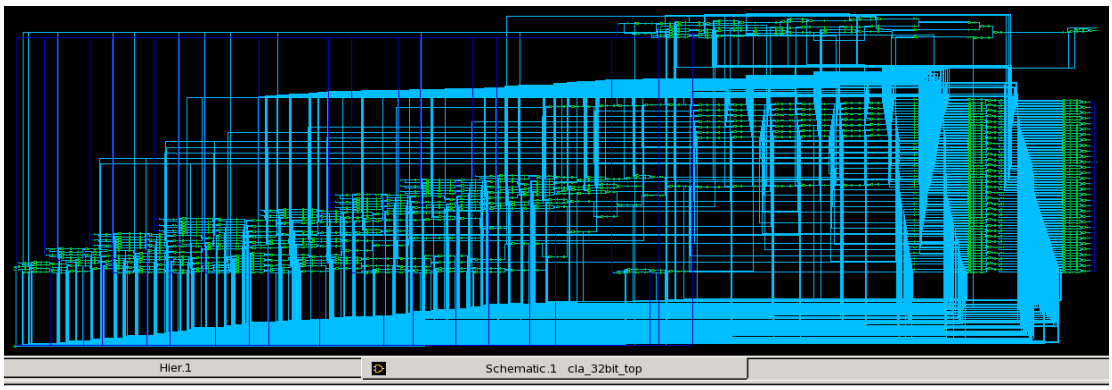

Figure 20: Gate level representation of a 32-bit Carry Lookahead Adder

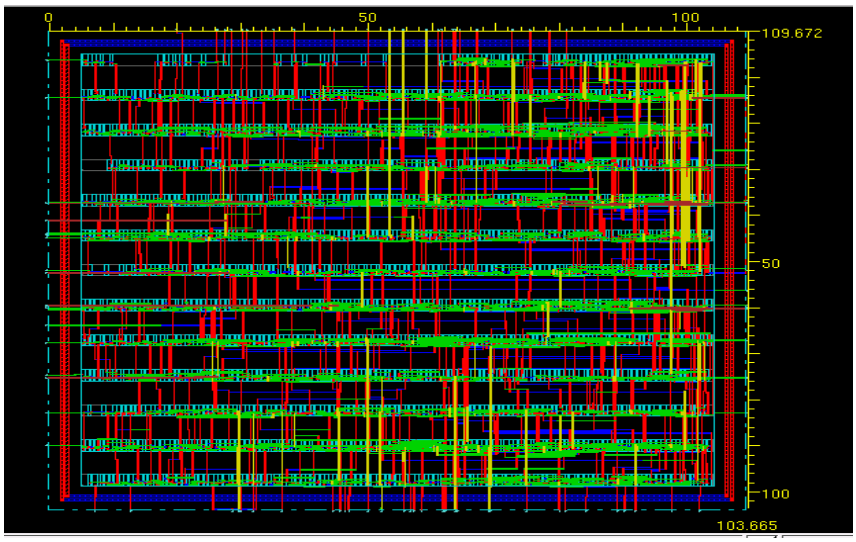

Figure 21: Layout of a 32-bit Carry Lookahead Adder

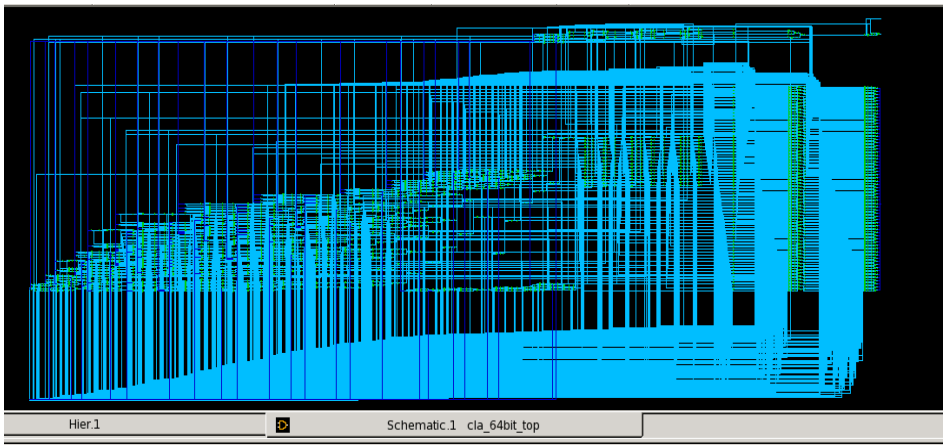

Figure 22: Gate level representation of a 64-bit Carry Lookahead Adder

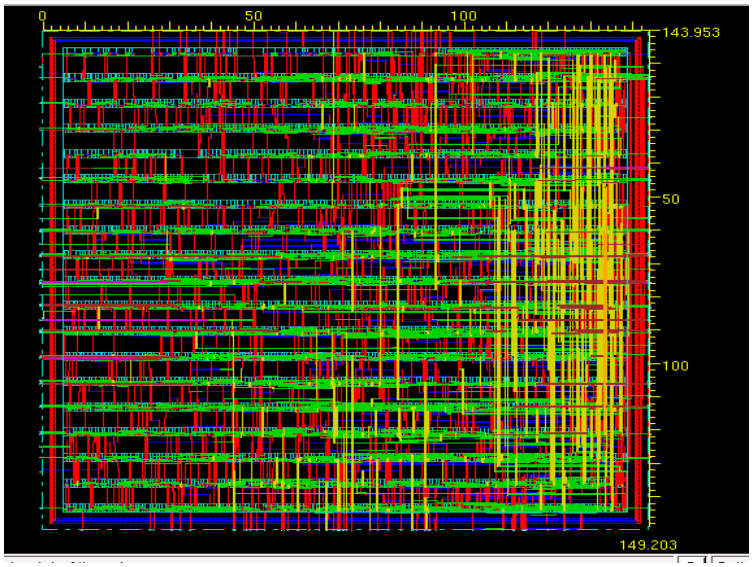

Figure 23: Layout of a 64-bit Carry Lookahead Adder

2.2. Schematic Level Design Entry

Cadence Virtuoso Schematic editor was used to make the schematic for the 16 bit, 32 bit and 64 bit parallel prefix adders. The standard cells used were 2-input NAND gates and INVERTER gates in 45 nm technology.

The schematic of all the parallel prefix adders consists of following three parts:

I. pg generation block

This block generates the propagate and generate signals using the following equations:

$$p_i = a_i \text{ xor } b_i$$

$$g_i = a_i \text{ and } b_i$$

The pg generation block for a single bit is as shown below.

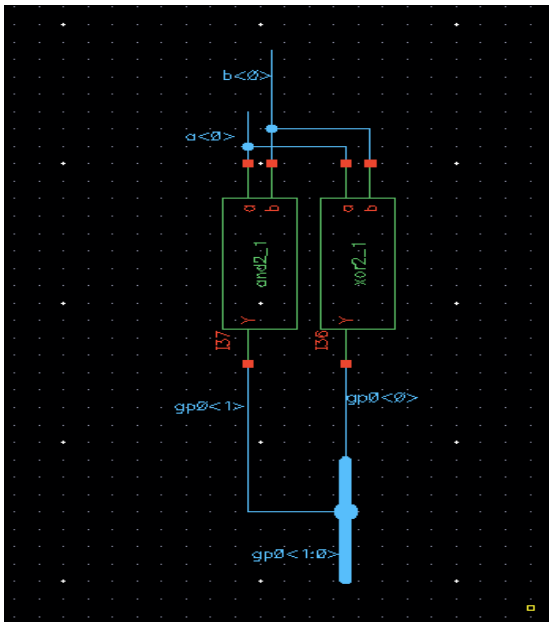

Figure 24: Pre processing block

II. Tree structure to compute PG

The prefix operator used to make the tree structure implements the following equations:

$$(g, p)\bullet(g', p') = (g + p.g', p.p')$$

The tree structure used for 16 bit adders is given below. It was similarly

extended to get the structure for 32 bit and 64 bit adders.

At the end of the tree, the group generate and propagate from bit 0 to bit i, i.e. G_i, P_i are available.

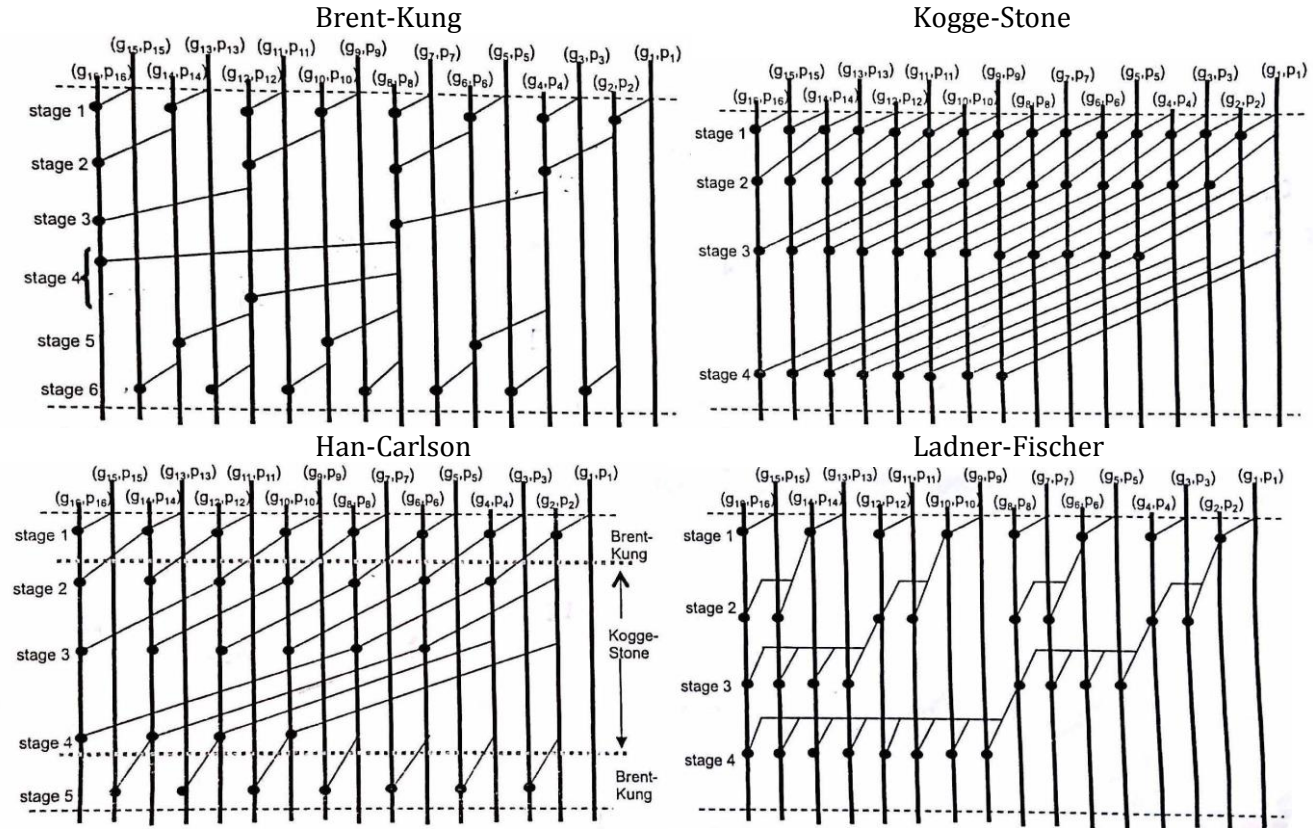

Figure 25: Tree Structure for 16 bit Parallel Prefix Adders

III. Sum and Carry Out bits from PG and C_{in}

This block generates the output sum bits using C_{in} and G_i, P_i with the following equations:

$$C_{-1} = C_{in}$$

$$C_i = G_i + P_i \,.\, C_{in}$$

$$S_i = p_i \text{ xor } C_{i-1}$$

The block for a single bit is as shown below.

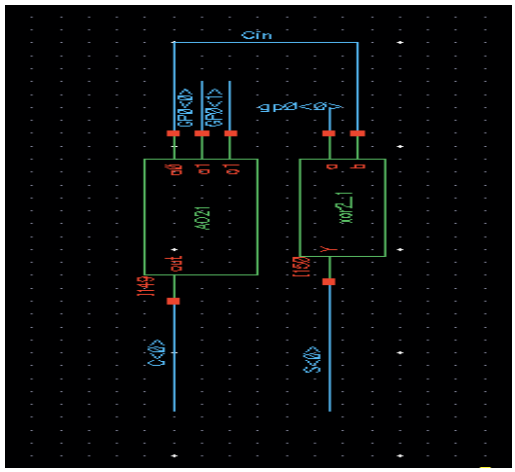

Figure 26: Post processing block

To reduce the overall delay of the adder, the critical path was studied and buffers were inserted to reduce the branching effort. Also high-fan-out nets were buffered to reduce the loading at each stage. The schematic and layout for various parallel prefix adders are shown below:

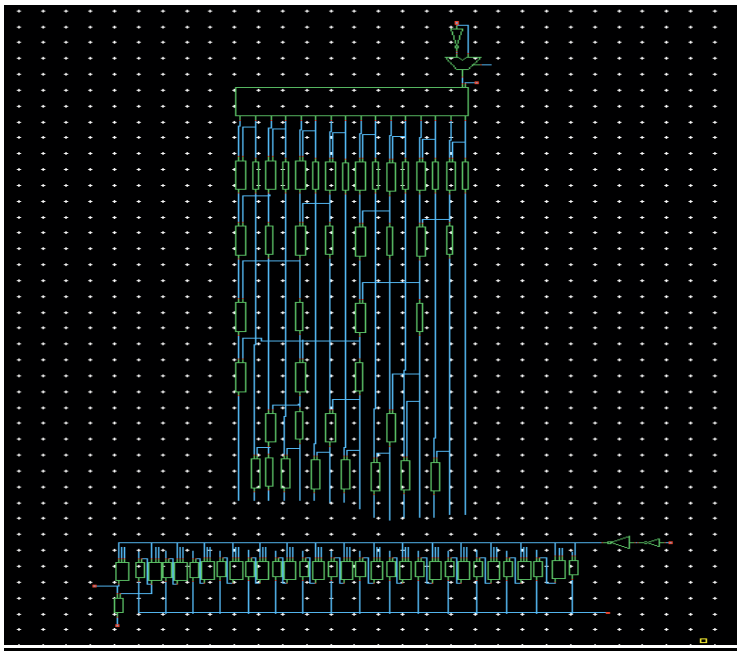

Figure 27: Schematic of a 16 bit Brent-Kung Adder

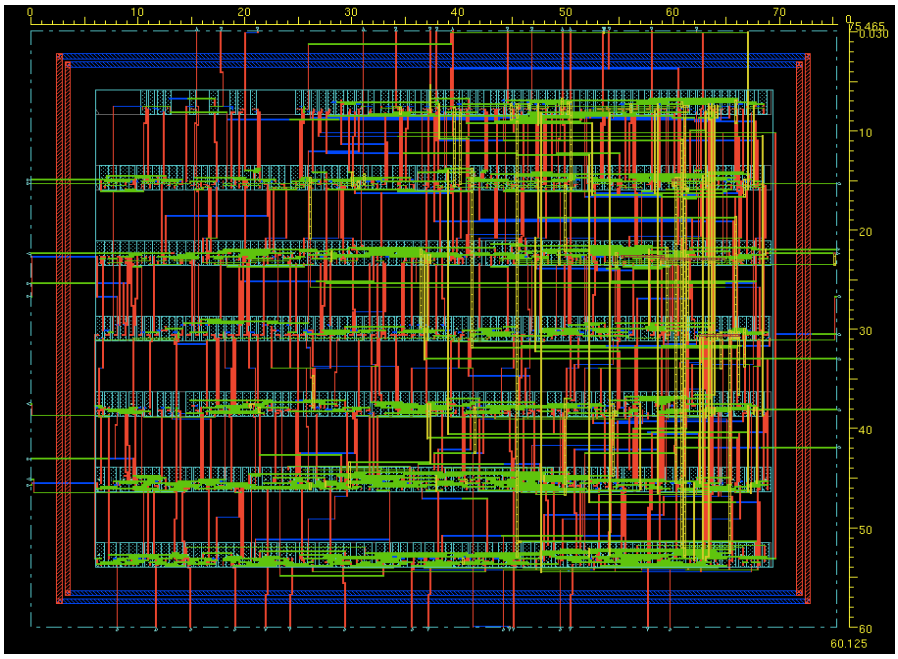

Figure 28: Layout of a 16-bit Brent Kung Adder

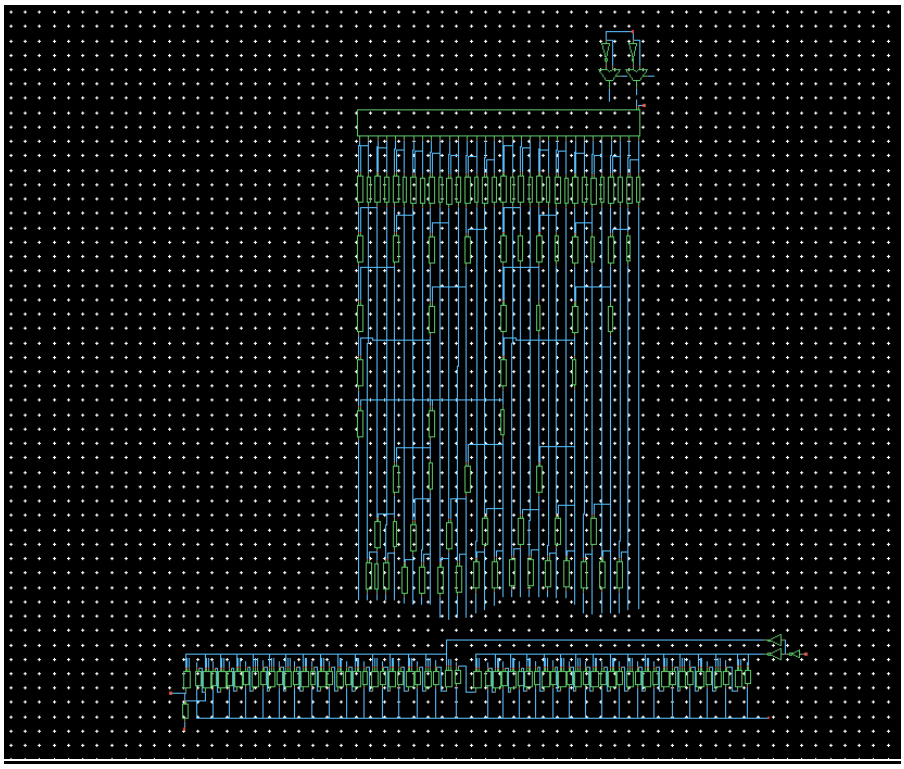

Figure 29: Schematic of a 32 bit Brent-Kung Adder

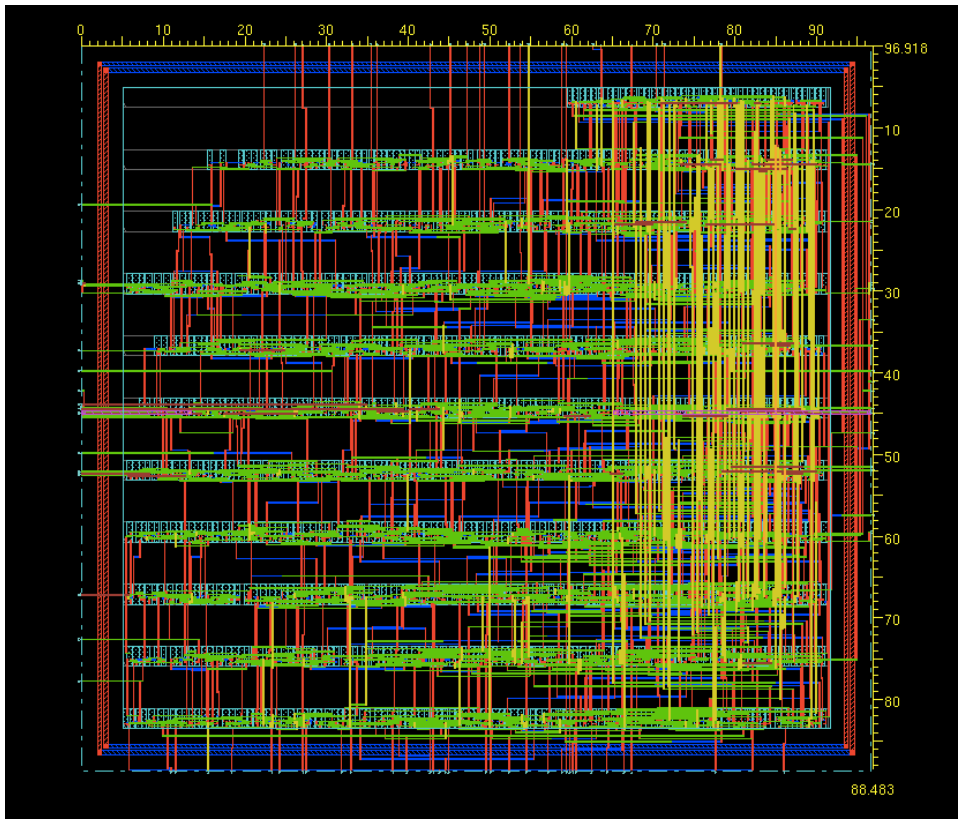

Figure 30: Layout of a 32-bit Brent Kung Adder

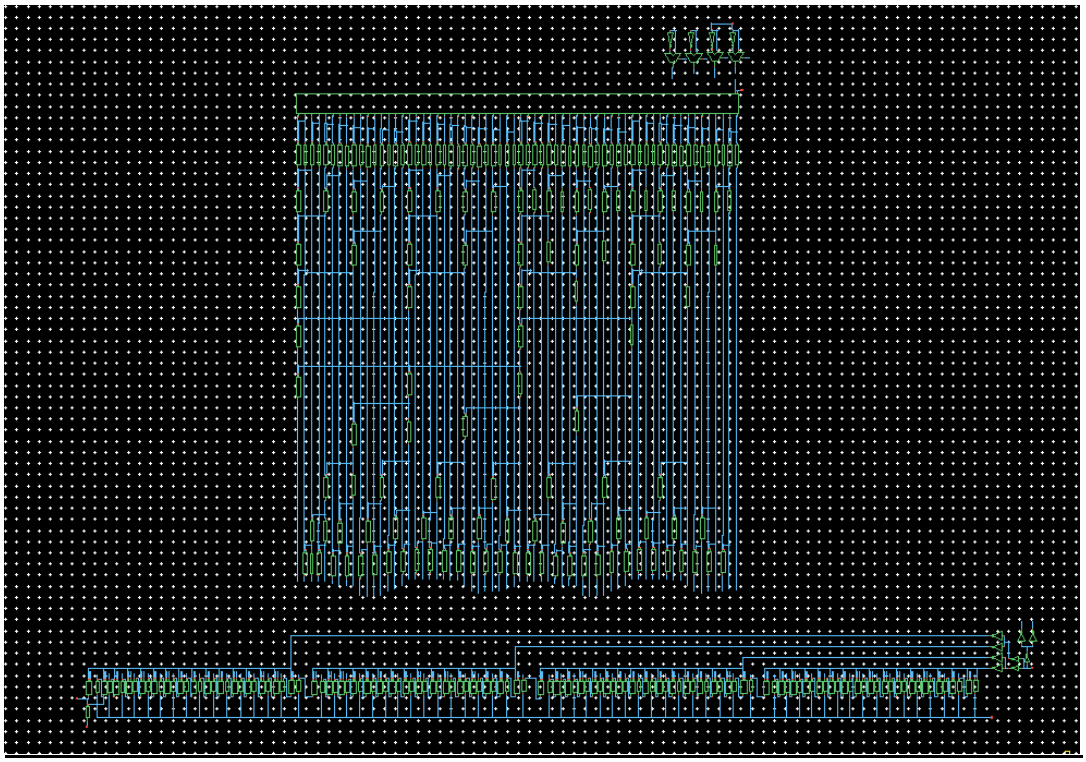

Figure 31: Schematic of a 64 bit Brent-Kung Adder

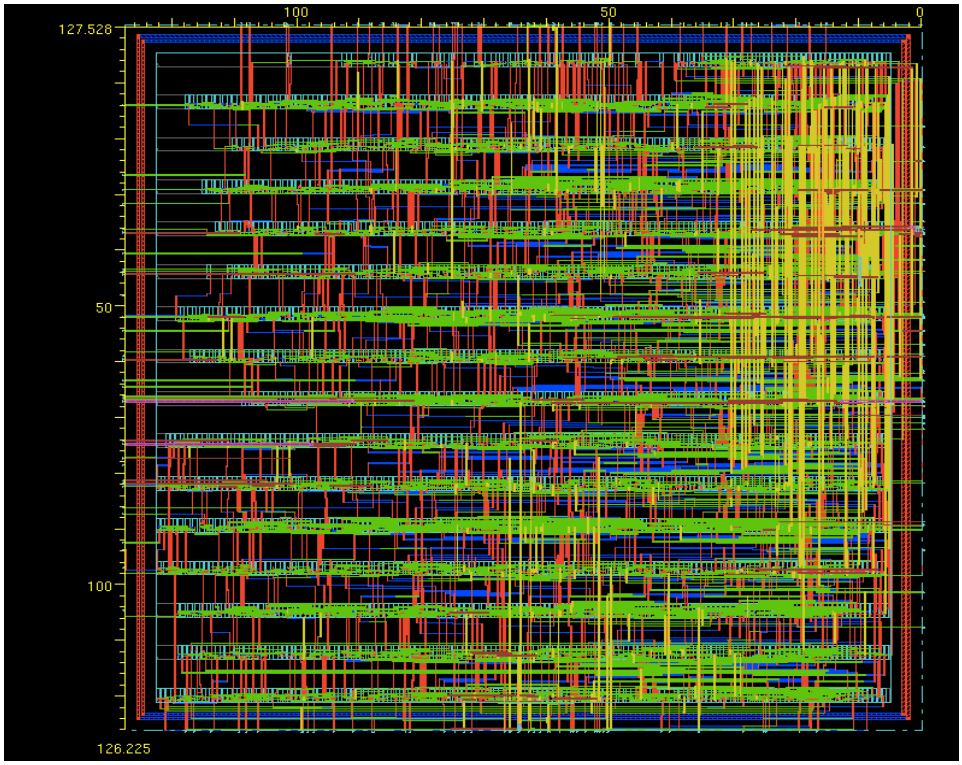

Figure 32: Layout of a 64-bit Brent Kung Adder

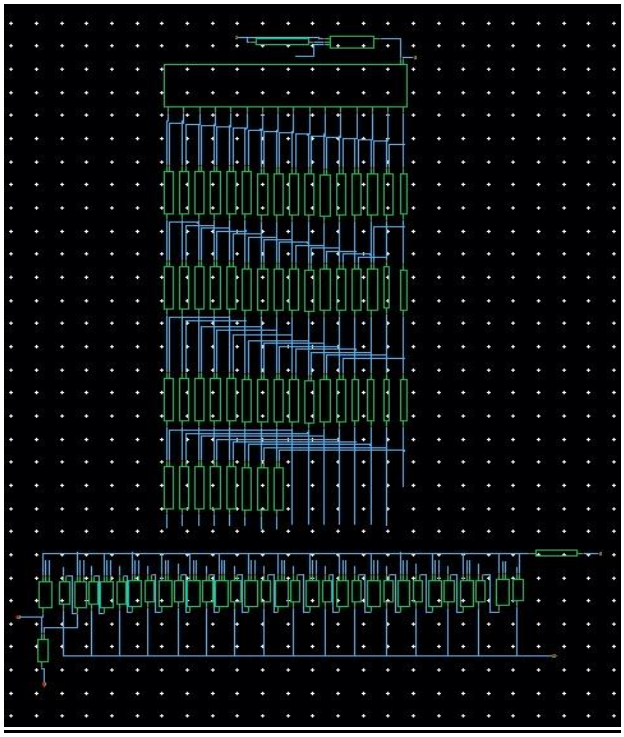

Figure 33: Schematic of a 16 bit Kogge-Stone Adder

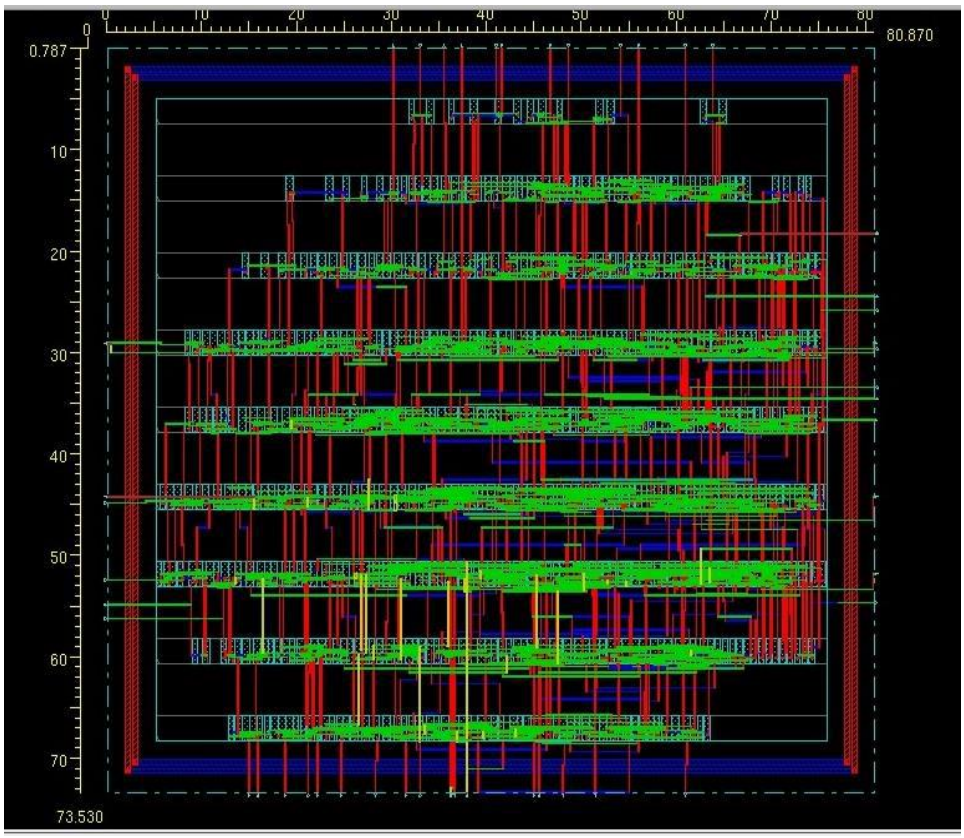

Figure 34: Layout of a 16 bit Kogge-Stone Adder

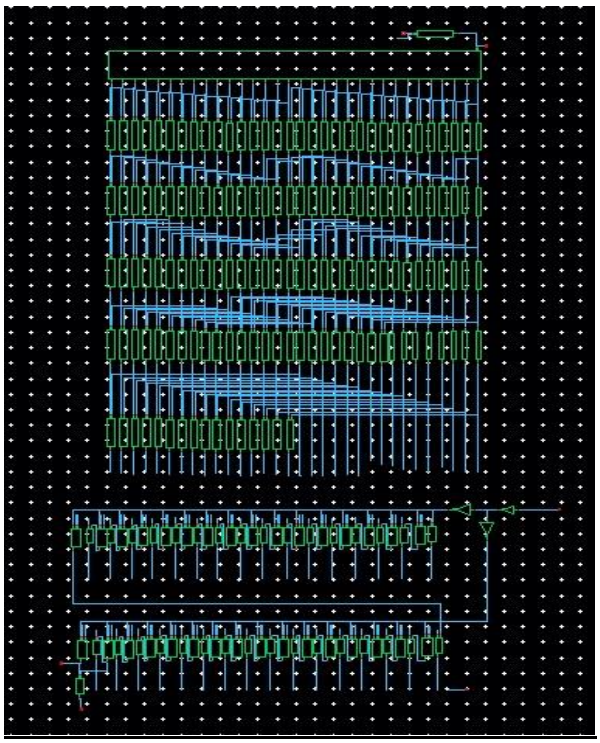

Figure 35: Schematic of a 32 bit Kogge-Stone Adder

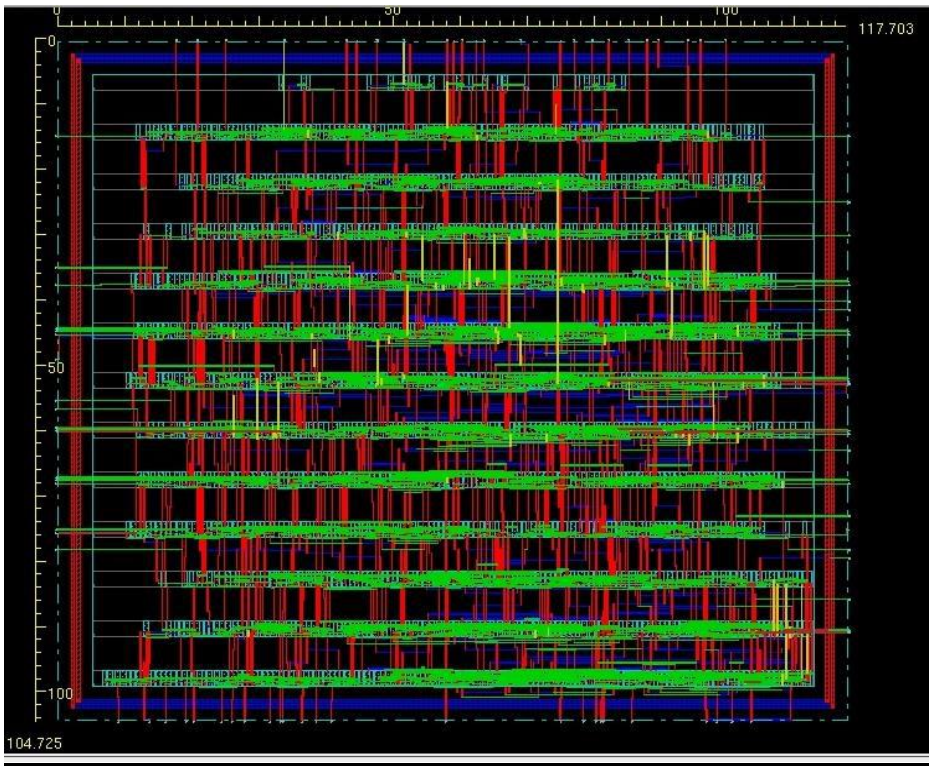

Figure 36: Layout of a 32 bit Kogge-Stone Adder

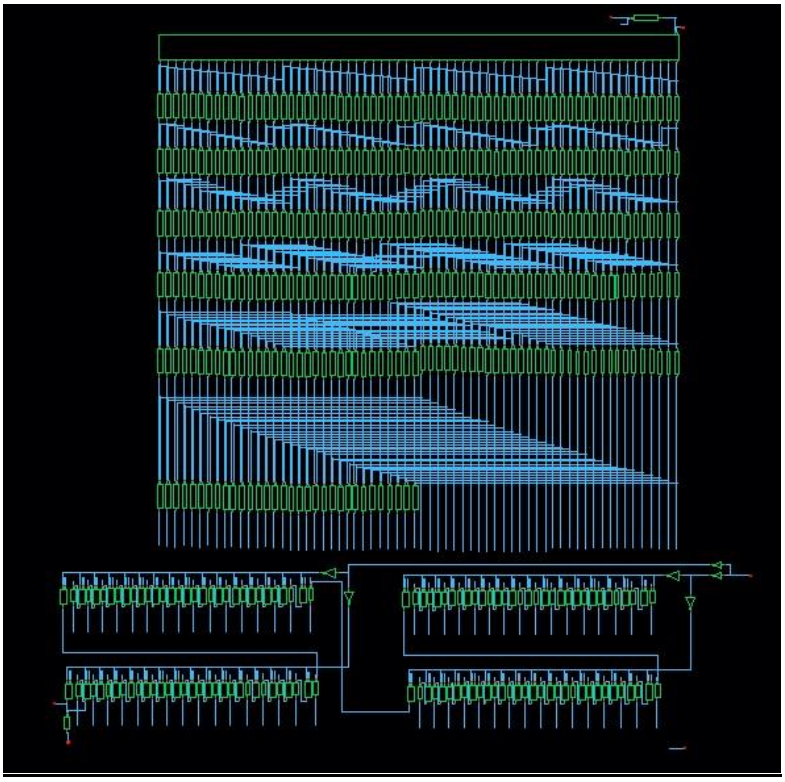

Figure 37: Schematic of a 64 bit Kogge-Stone Adder

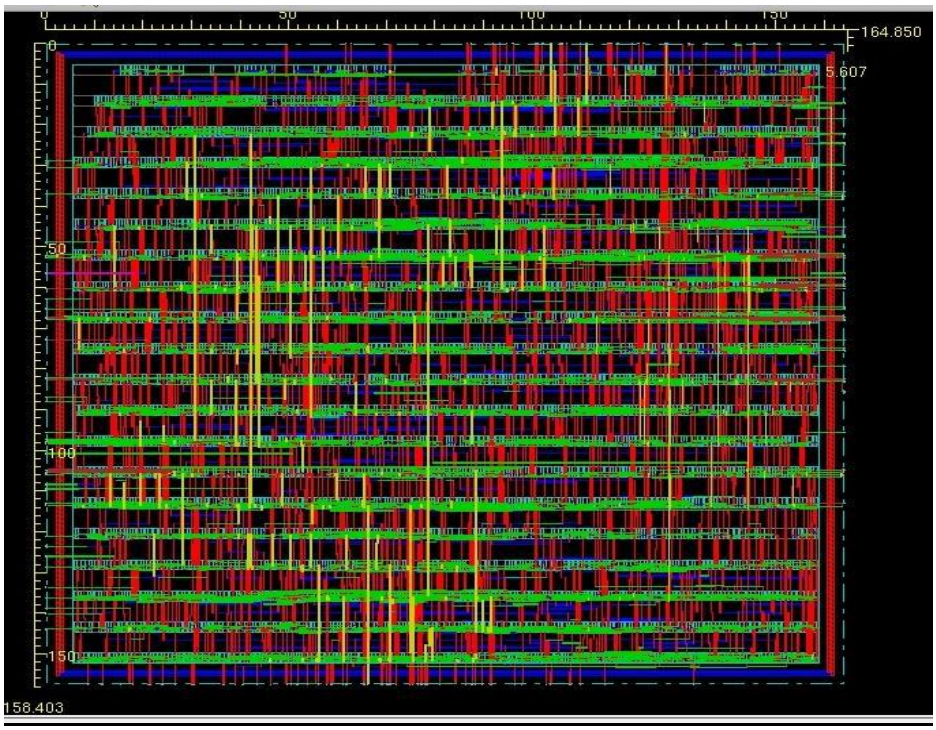

Figure 38: Layout of a 64 bit Kogge-Stone Adder

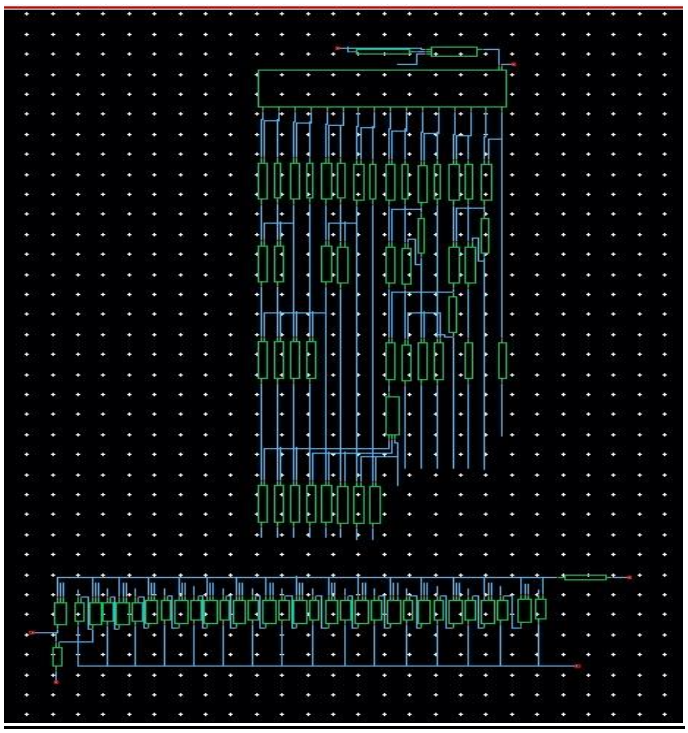

Figure 39: Schematic of a 16 bit Ladner-Fischer Adder

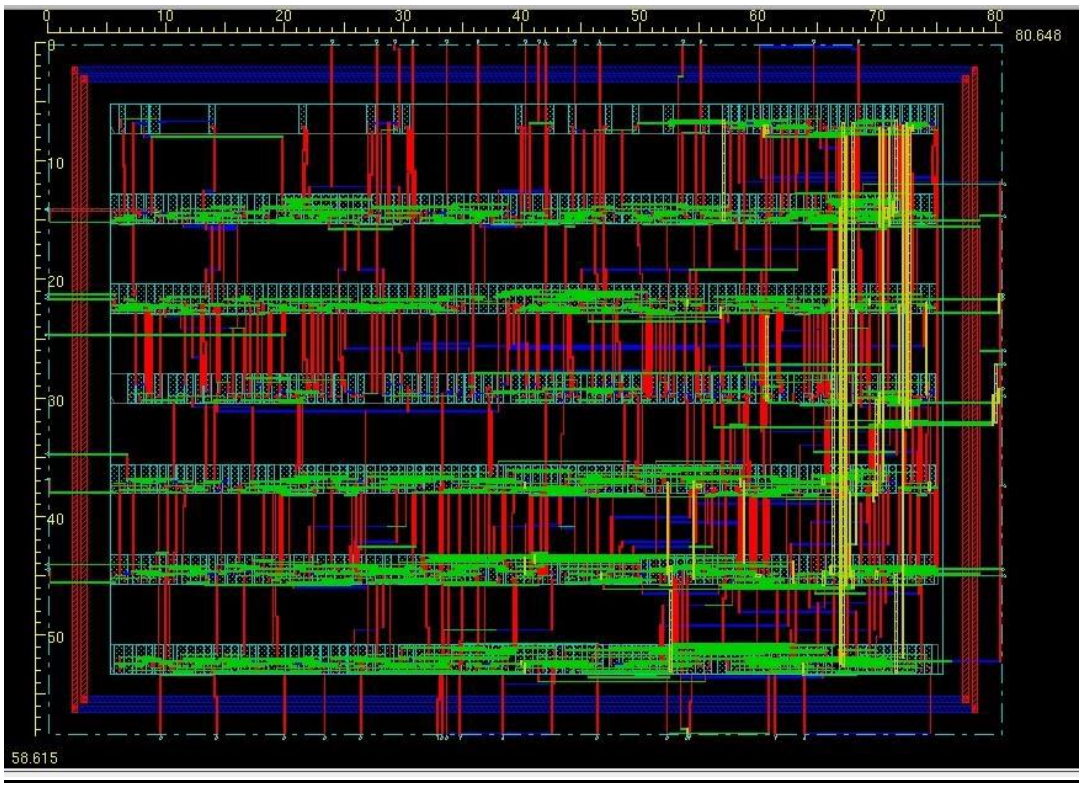

Figure 40: Layout of a 16 bit Ladner-Fischer Adder

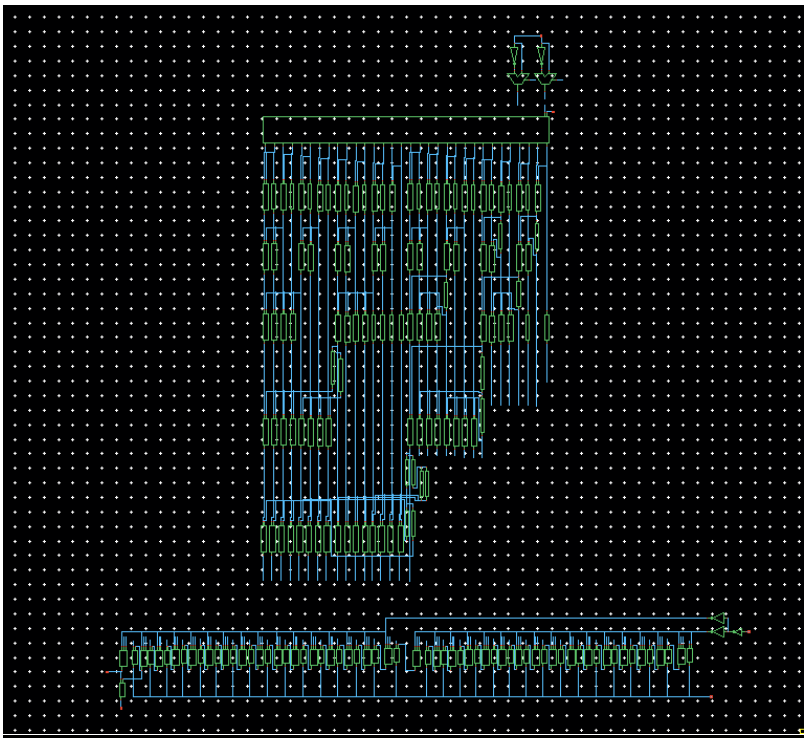

Figure 41: Schematic of a 32 bit Ladner-Fischer Adder

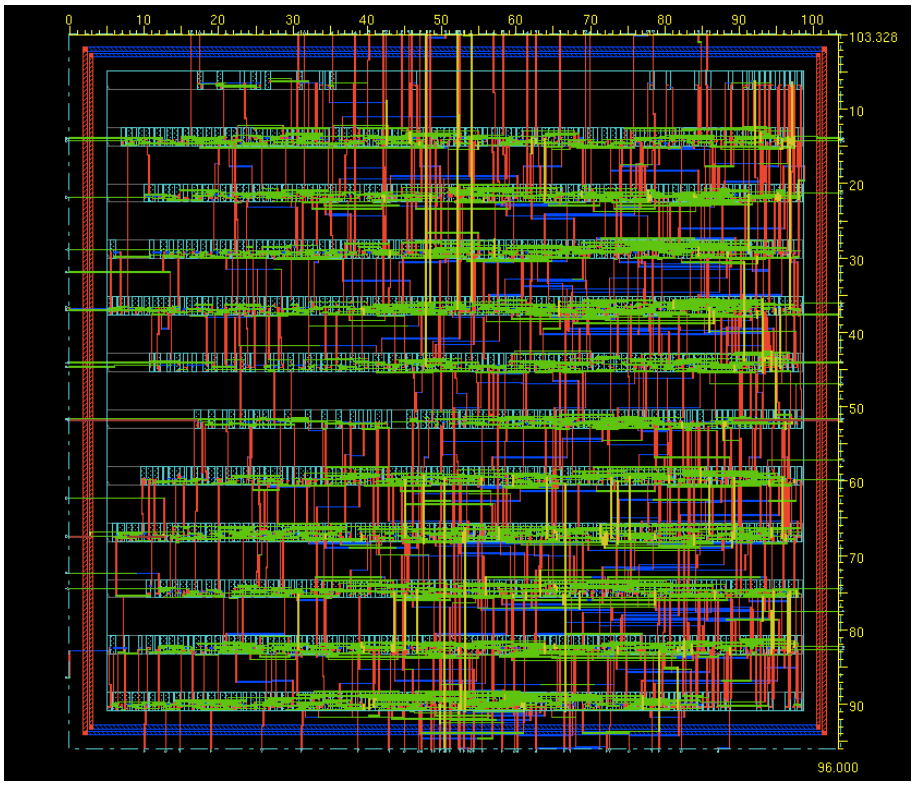

Figure 42: Layout of a 32 bit Ladner-Fischer Adder

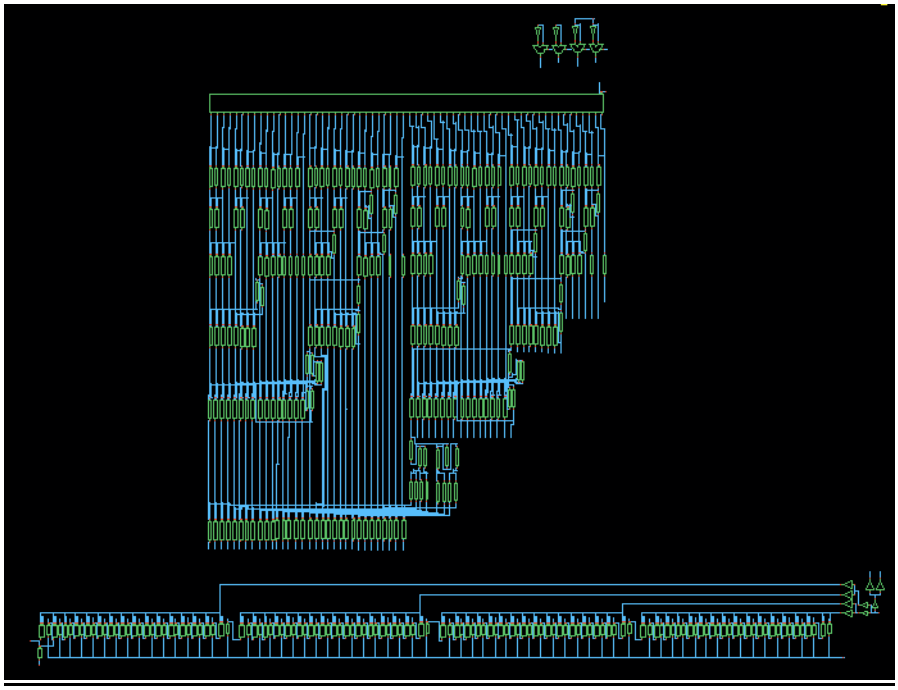

Figure 43: Schematic of a 64 bit Ladner-Fischer Adder

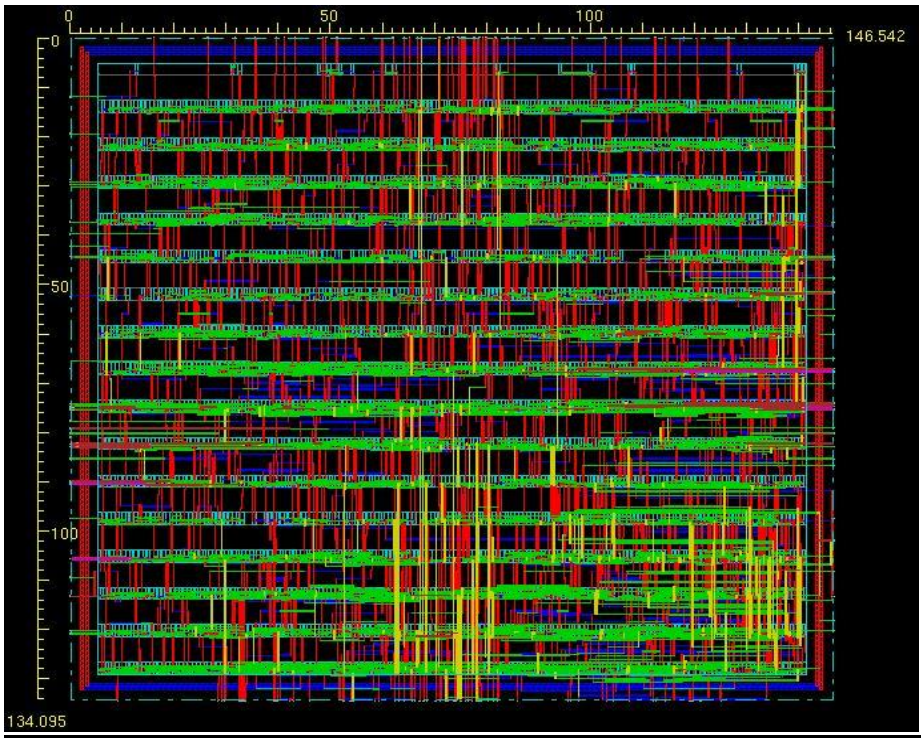

Figure 44: Layout of a 64 bit Ladner-Fischer Adder

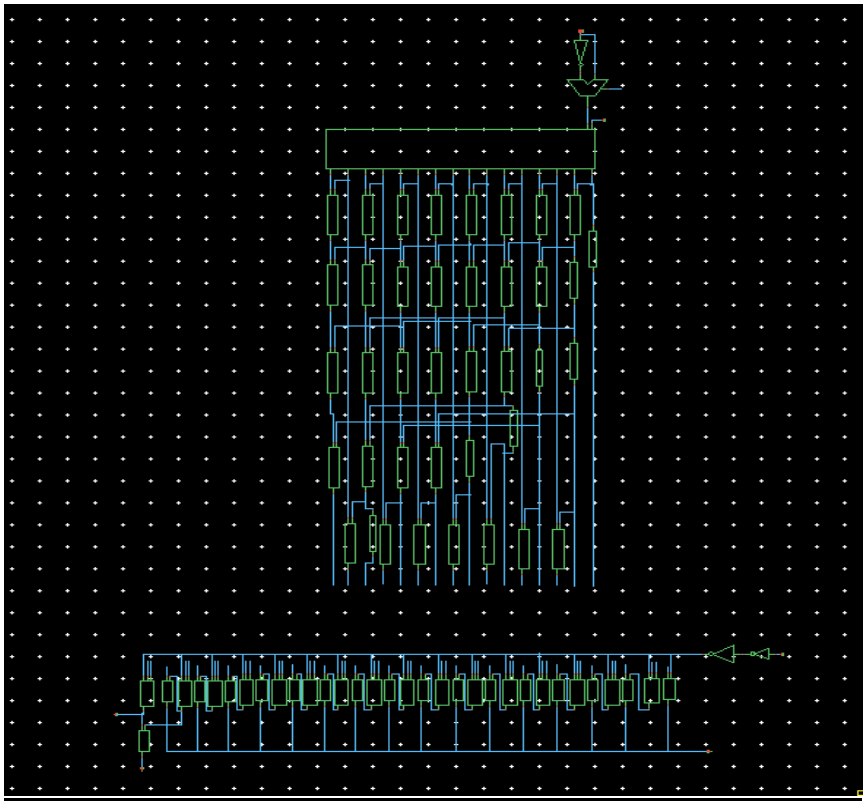

Figure 45: Schematic of a 16 bit Han-Carlson Adder

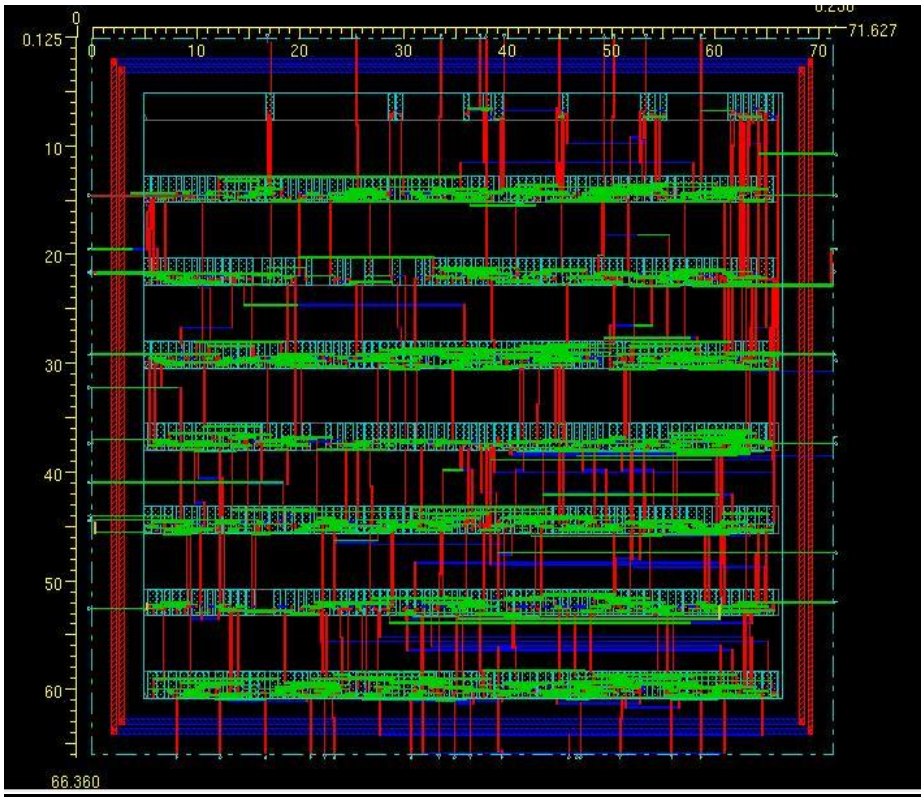

Figure 46: Layout of a 16 bit Han-Carlson Adder

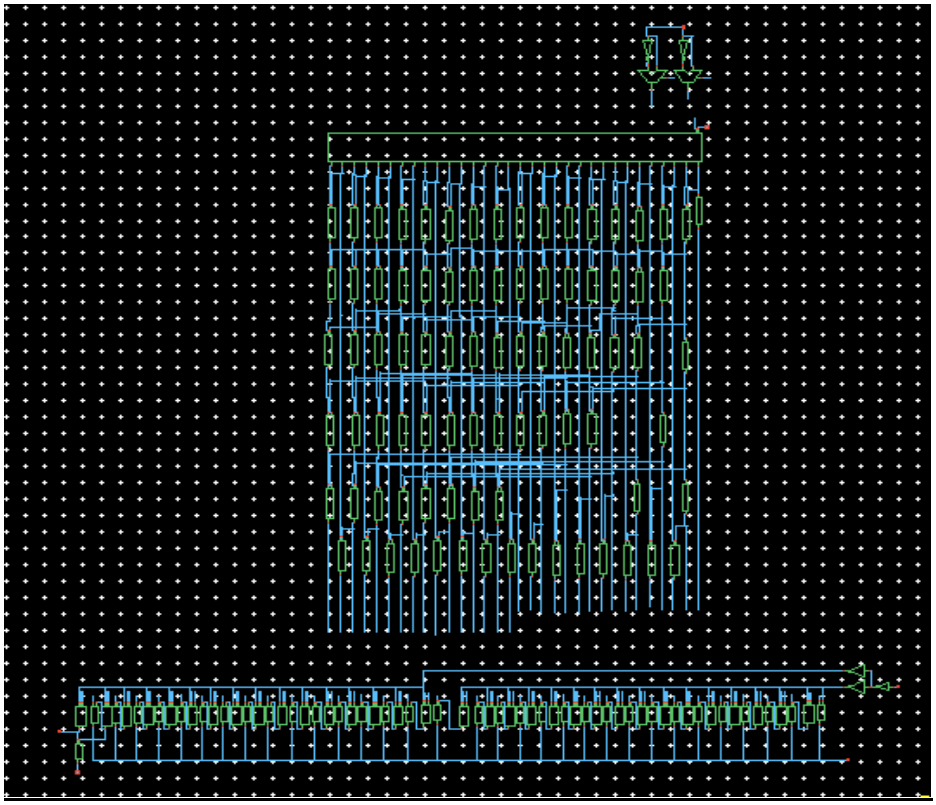

Figure 47: Schematic of a 32 bit Han-Carlson Adder

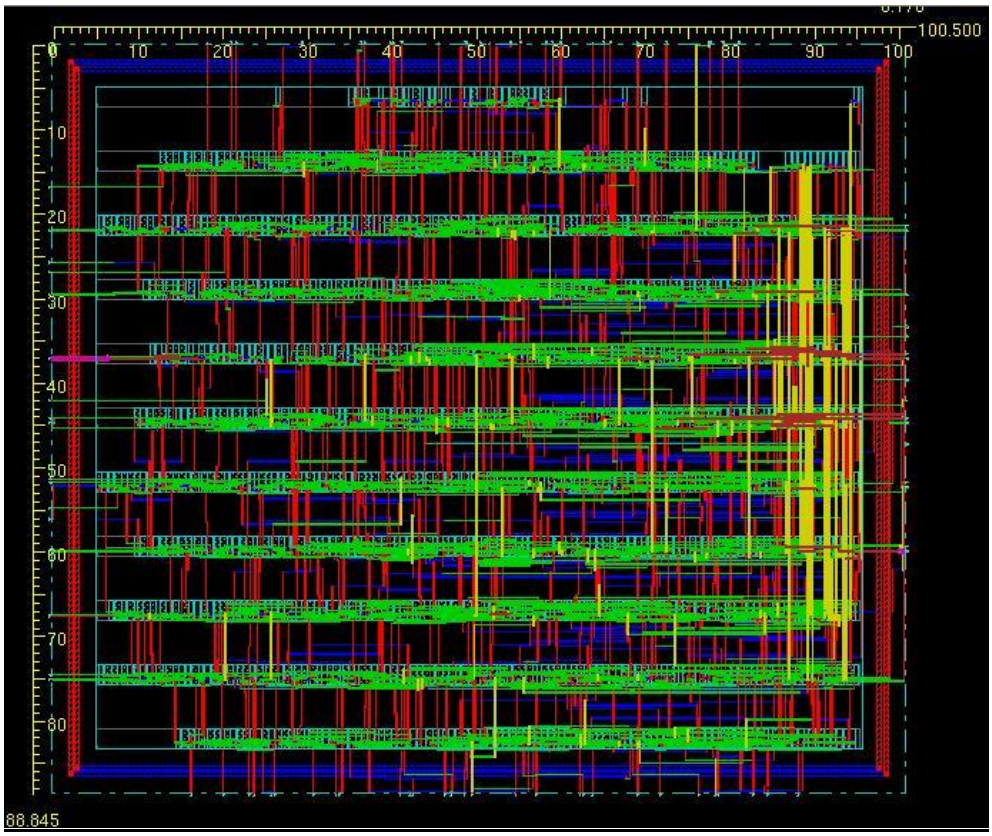

Figure 48: Layout of a 32 bit Han-Carlson Adder

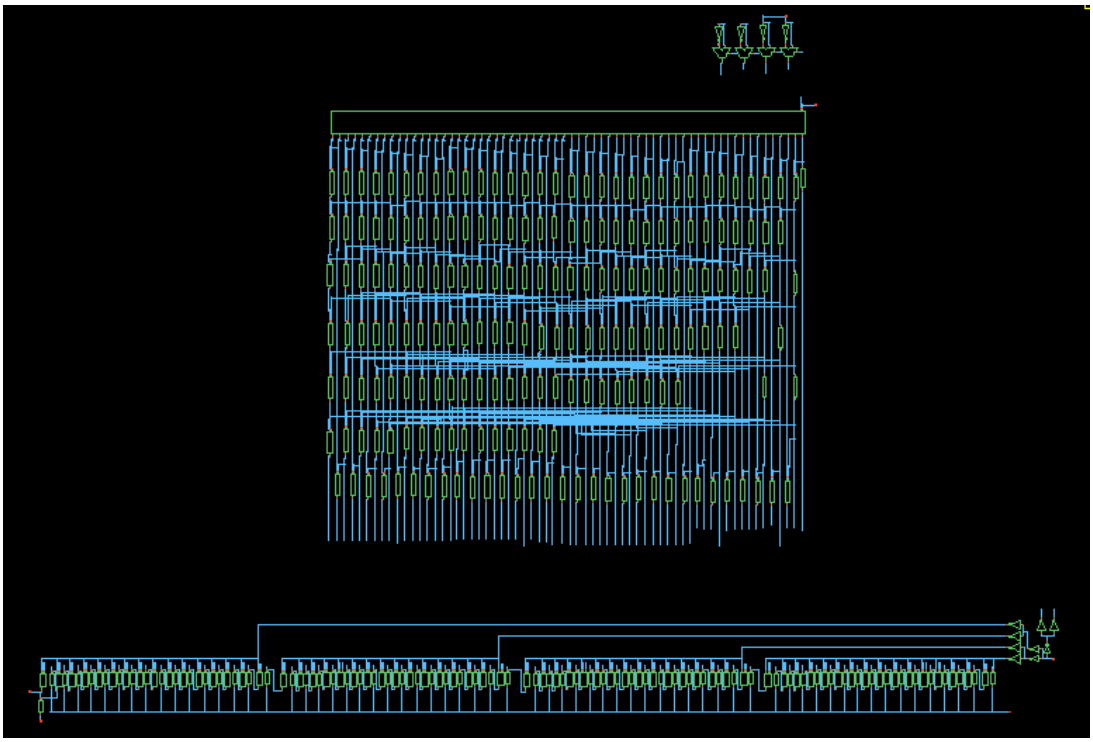

Figure 49: Schematic of a 64 bit Han-Carlson Adder

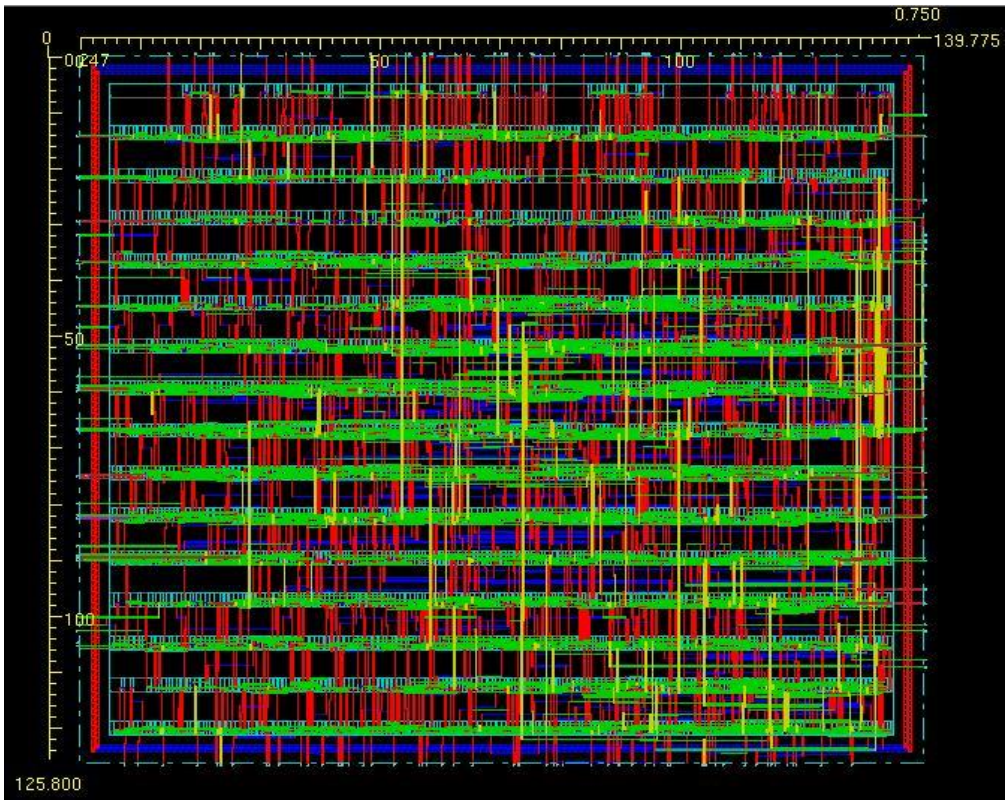

Figure 50: Layout of a 64 bit Han-Carlson Adder

2.3. Functional Verification

Verilog XL simulations were performed to verify the functional correctness of the adder. The waveforms for 16 bit, 32 bit and 64 bit adders are shown below.

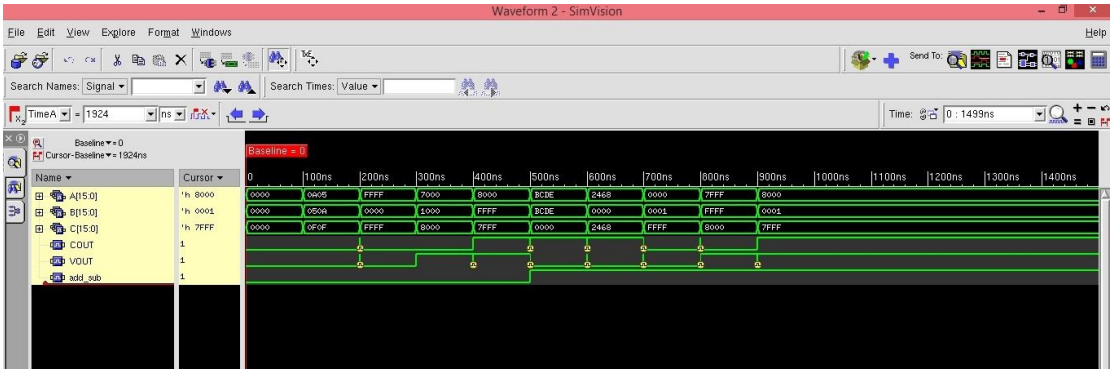

Figure 51: Waveform for 16 bit adders

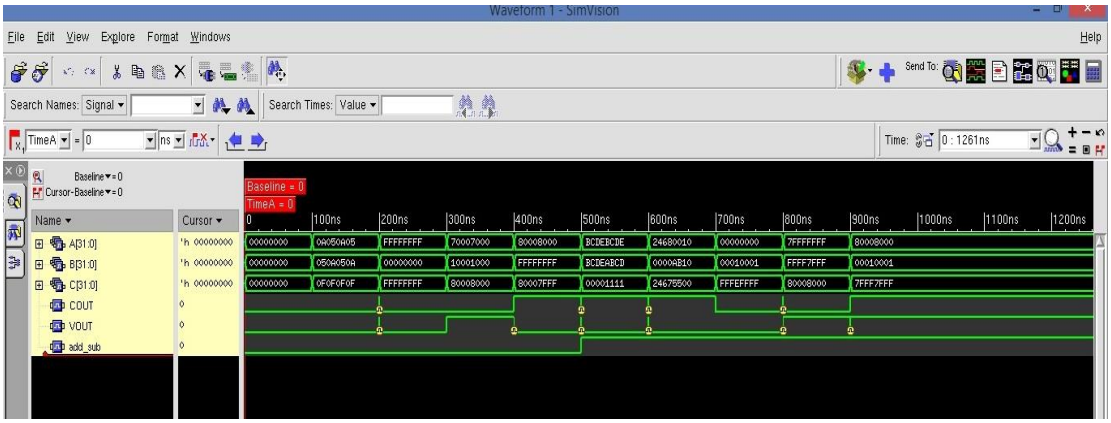

Figure 52: Waveform for 32 bit adders

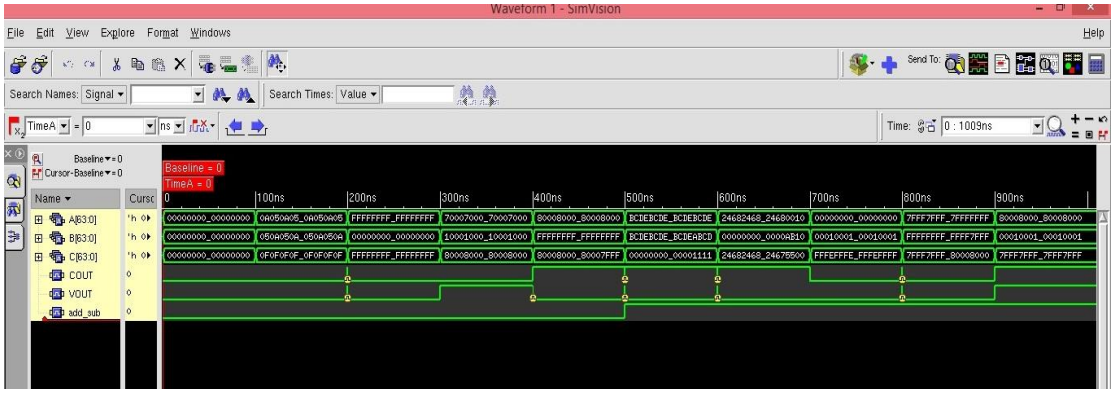

Figure 53: Waveform for 64 bit adders

After functional verification, Synopsys PrimeTime tool was used to measure the Pre APR as well as Post APR delay of the adders. The timing report for the pre-apr netlist of 16 bit carry lookahead adder is shown here. The unit used is nanoseconds.

```
Startpoint: add_sub (input port clocked by vclk)
Endpoint: s[15] (output port)
Path Group: (none)
Path Type: max

Point                                                    Incr       Path
-----------------------------------------------------------------------------
input external delay                                   0.0000     0.0000 f
add_sub (in)                                           0.0000     0.0000 f
U522/Y (INVX1)                                         0.1020     0.1020 r
U53/Y (NAND2X1)                                        0.0425     0.1445 f
U234/Y (INVX1)                                        -0.0022     0.1424 r
U235/Y (INVX1)                                         0.0125     0.1549 f
U52/Y (NAND2X1)                                        0.0109     0.1658 r
U436/Y (INVX1)                                         0.0325     0.1983 f
cla16_inst/bits3_0/bit1/U11/Y (NAND2X1)                0.0292     0.2274 r
U498/Y (INVX1)                                         0.0199     0.2473 f
U499/Y (INVX1)                                         0.0133     0.2606 r
cla16_inst/bits3_0/lookahead/U13/Y (NAND2X1)           0.0118     0.2724 f
U84/Y (INVX1)                                          0.0030     0.2754 r
U85/Y (INVX1)                                          0.0125     0.2879 f
cla16_inst/bits3_0/lookahead/U12/Y (NAND2X1)           0.0109     0.2988 r
U82/Y (INVX1)                                          0.0201     0.3189 f
U83/Y (INVX1)                                         -0.0004     0.3185 r
cla16_inst/bits3_0/lookahead/U11/Y (NAND2X1)           0.0118     0.3303 f
U80/Y (INVX1)                                          0.0030     0.3333 r
U81/Y (INVX1)                                          0.0125     0.3458 f
cla16_inst/bits3_0/lookahead/U10/Y (NAND2X1)           0.0109     0.3567 r
U78/Y (INVX1)                                          0.0201     0.3768 f
U79/Y (INVX1)                                         -0.0004     0.3764 r
cla16_inst/bits3_0/lookahead/U9/Y (NAND2X1)            0.0118     0.3882 f
U76/Y (INVX1)                                          0.0030     0.3912 r
U77/Y (INVX1)                                          0.0125     0.4037 f
cla16_inst/bits3_0/lookahead/U8/Y (NAND2X1)            0.0109     0.4146 r
U374/Y (INVX1)                                         0.0240     0.4386 f
cla16_inst/cla16_12_lookahead/U18/Y (NAND2X1)          0.0253     0.4638 r
U516/Y (INVX1)                                         0.0241     0.4879 f
U517/Y (INVX1)                                         0.0153     0.5032 r
cla16_inst/cla16_12_lookahead/U17/Y (NAND2X1)          0.0118     0.5151 f
U216/Y (INVX1)                                         0.0030     0.5180 r
U217/Y (INVX1)                                         0.0125     0.5306 f
cla16_inst/cla16_12_lookahead/U16/Y (NAND2X1)          0.0109     0.5415 r
U514/Y (INVX1)                                         0.0241     0.5656 f
U515/Y (INVX1)                                         0.0153     0.5809 r
cla16_inst/cla16_12_lookahead/U15/Y (NAND2X1)          0.0118     0.5927 f
U214/Y (INVX1)                                         0.0030     0.5957 r
U215/Y (INVX1)                                         0.0125     0.6082 f
cla16_inst/cla16_12_lookahead/U14/Y (NAND2X1)          0.0109     0.6191 r
U492/Y (INVX1)                                         0.0241     0.6432 f
U493/Y (INVX1)                                         0.0087     0.6519 r
cla16_inst/bits15_12/lookahead/U19/Y (NAND2X1)         0.0089     0.6608 f
U138/Y (INVX1)                                         0.0030     0.6638 r
U139/Y (INVX1)                                         0.0125     0.6763 f
cla16_inst/bits15_12/lookahead/U18/Y (NAND2X1)         0.0109     0.6872 r
U490/Y (INVX1)                                         0.0241     0.7113 f
U491/Y (INVX1)                                         0.0085     0.7198 r
cla16_inst/bits15_12/lookahead/U17/Y (NAND2X1)         0.0118     0.7316 f
U136/Y (INVX1)                                         0.0030     0.7346 r
U137/Y (INVX1)                                         0.0125     0.7471 f
cla16_inst/bits15_12/lookahead/U16/Y (NAND2X1)         0.0109     0.7580 r
U488/Y (INVX1)                                         0.0241     0.7821 f
U489/Y (INVX1)                                         0.0085     0.7905 r
cla16_inst/bits15_12/lookahead/U15/Y (NAND2X1)         0.0118     0.8024 f
U134/Y (INVX1)                                         0.0030     0.8053 r
U135/Y (INVX1)                                         0.0125     0.8178 f
cla16_inst/bits15_12/lookahead/U14/Y (NAND2X1)         0.0109     0.8287 r
U518/Y (INVX1)                                         0.0241     0.8529 f
U519/Y (INVX1)                                         0.0157     0.8686 r
cla16_inst/bits15_12/bit3/U9/Y (NAND2X1)               0.0089     0.8775 f
U198/Y (INVX1)                                         0.0027     0.8802 r
U199/Y (INVX1)                                         0.0125     0.8927 f
cla16_inst/bits15_12/bit3/U8/Y (NAND2X1)               0.0109     0.9036 r
U416/Y (INVX1)                                         0.0240     0.9276 f
cla16_inst/bits15_12/bit3/U6/Y (NAND2X1)               0.0253     0.9528 r
U196/Y (INVX1)                                         0.0203     0.9732 f
U197/Y (INVX1)                                        -0.0005     0.9727 r
cla16_inst/bits15_12/bit3/U5/Y (NAND2X1)               0.0093     0.9820 f
s[15] (out)                                            0.0000     0.9820 f
data arrival time                                                 0.9820
-----------------------------------------------------------------------------
```

Figure 54: Timing report for 16 bit Carry Lookahead Adder

For power analysis, PT-PX (by Synopsys) was used. Average power analysis was performed by manually annotating the switching activity using "set_switching_activity" command.

A toggle rate of 0.25 was specified for data nets i.e. 0.25 toggles per period of the base clock. Static probability was specified as 0.25 which means that the nets are on logic state 1 for 25 % of the time per period of the base clock.

```
Information: Running averaged power analysis... (PWR-601)
****************************************
Report : Averaged Power
Design : cla_16bit_top
Version: H-2012.12
Date   : Sun Oct 11 19:37:59 2015
****************************************


  Attributes
  ----------
     i  -  Including register clock pin internal power
     u  -  User defined power group

                    Internal  Switching  Leakage    Total
Power Group         Power     Power      Power      Power   (     %)  Attrs
--------------------------------------------------------------------------------
clock_network          0.0000    0.0000    0.0000    0.0000 ( 0.00%)  i
register               0.0000    0.0000    0.0000    0.0000 ( 0.00%)
combinational       1.062e-05 2.053e-05 1.719e-06 3.287e-05 (100.00%)
sequential             0.0000    0.0000    0.0000    0.0000 ( 0.00%)
memory                 0.0000    0.0000    0.0000    0.0000 ( 0.00%)
io_pad                 0.0000    0.0000    0.0000    0.0000 ( 0.00%)
black_box              0.0000    0.0000    0.0000    0.0000 ( 0.00%)

  Net Switching Power  = 2.053e-05   (62.46%)
  Cell Internal Power  = 1.062e-05   (32.31%)
  Cell Leakage Power   = 1.719e-06   ( 5.23%)
                         ---------
Total Power            = 3.287e-05  (100.00%)
```

Figure 55: Power report for 16 bit Carry Lookahead Adder

3. Results

The tables and graphs comparing Pre APR and Post APR delays are as given below. It can be seen that ripple carry adders have the worst delay among all the adders. Carry select and carry lookahead offers significant improvement over the ripple carry adder. The delay of carry select adder is more since it is made up of blocks of ripple carry adders and the carry has to propagate sequentially within a block. Parallel prefix adders offer the minimum delay as it is proportional to (Log N) where N is the number of bits in the operand. Among them, Brent-Kung has the worst delay among all the four adders since it has more stages compared to the other adders. For 16 bit, Kogge-Stone, Han-Carlson and Ladner-Fischer have approximately the same delay since the fanout is not an important factor for operand size of 16. For 32 and 64 bits, Kogge-Stone has the least delay since the fanout at any stage is limited to two and hence the delay is less. Han-Carlson and Ladner-Fischer have approximately the same delay even though Han-Carlson adder has one more stage than Ladner-Fischer. This is because Ladner-Fischer adders have large fanouts, due to which the delays of both the adders are comparable even after inserting buffers in Ladner-Fischer.

Table 1: Pre APR Delay (in ns)

	16 Bit	**32 Bit**	**64 Bit**
Ripple Carry	1.6753	3.1096	5.743
Carry Select	1.0028	1.4236	2.2239
Carry Lookahead	0.9933	1.3947	1.4832
Brent-Kung	0.5568	0.6606	0.8183
Ladner-Fischer	0.5125	0.5882	0.7261
Han-Carlson	0.5117	0.574	0.6877
Kogge-Stone	0.4959	0.5496	0.6597

Note: Due to large delay of ripple carry adders, it is omitted from the graph so as to analyze the trend in the delay of other adders

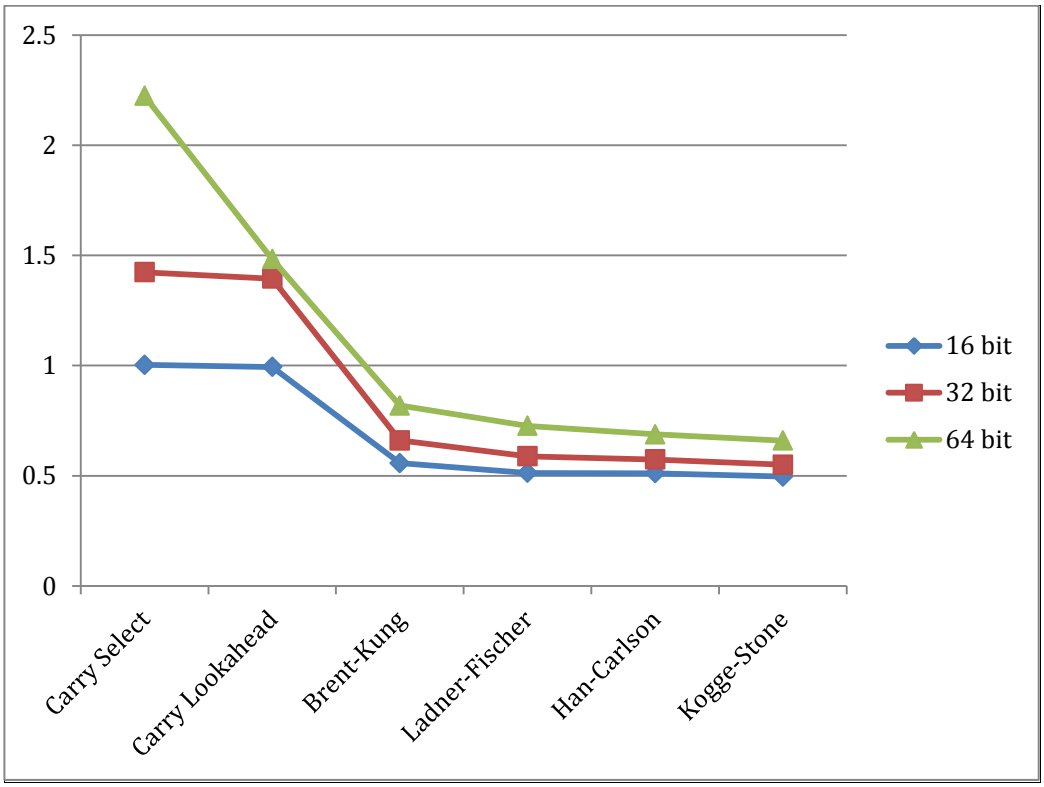

Figure 56: Pre APR Delay (in ns)

Table 2: Post APR Delay (in ns)

	16 Bit	**32 Bit**	**64 Bit**
Ripple Carry	2.2987	4.2099	9.545
Carry Select	1.6825	2.2043	4.1861
Carry Lookahead	1.5346	1.9725	2.202
Brent-Kung	0.7493	1.0528	1.3653
Ladner-Fischer	0.6671	0.8042	1.0805
Han-Carlson	0.664	0.8546	1.047
Kogge-Stone	0.6572	0.7803	0.971

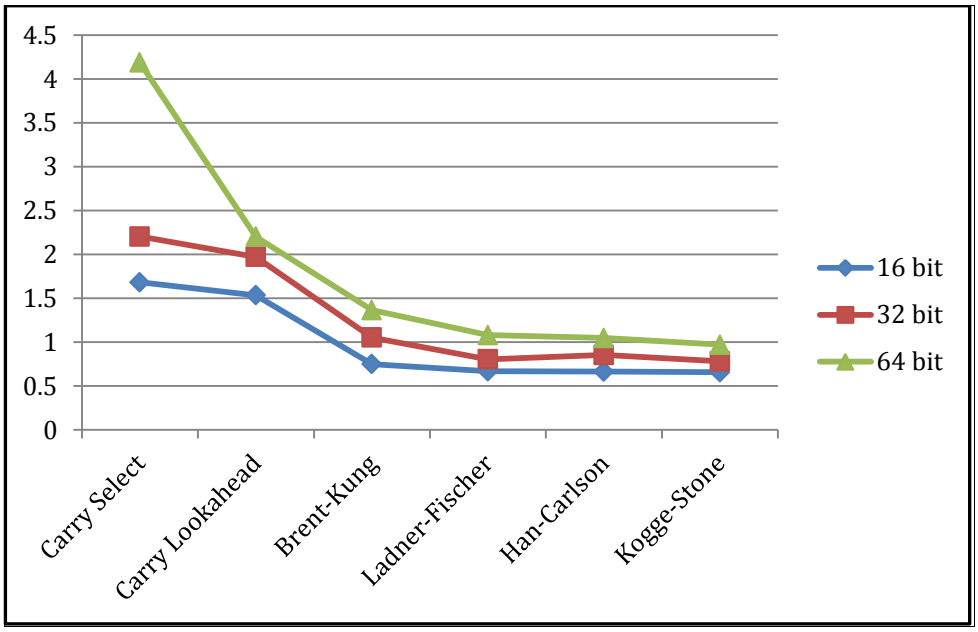

Figure 57: Post APR Delay (in ns)

The tables and graphs comparing Gate Count and Area of the Layout are given below. It is evident from these that ripple carry adders are the simplest ones to implement. It doesn't have any fanout issues and is regular in structure. In contrast, carry select adders need a large number of gates since, it has two blocks of ripple carry adders performing the addition and a multiplexer to choose the correct output. The gate count of carry lookahead adder is a little high since it needs gates with up to 4 inputs. Since only 2 input NAND and NOR gates were used, the complexity increased. Among the tree adders, Brent-Kung adder has the least complexity whereas Kogge-Stone has the most complexity. This is expected since Kogge-stone adder has a large number of prefix operations. Complexity of Han-Carlson is somewhat in between Brent-Kung and Kogge-Stone. Ladner-Fischer has a comparatively lesser number of gates but it has high fanout as the number of bits in the operands increase.

Table 3: Gate count results

	16 Bit	32 Bit	64 Bit
Ripple Carry	588	1166	2326
Carry Select	832	1764	3528
Carry Lookahead	754	1507	3020
Brent-Kung	655	1297	2616
Han-Carlson	641	1325	2813
Ladner-Fischer	676	1458	3143
Kogge-Stone	744	1670	3718

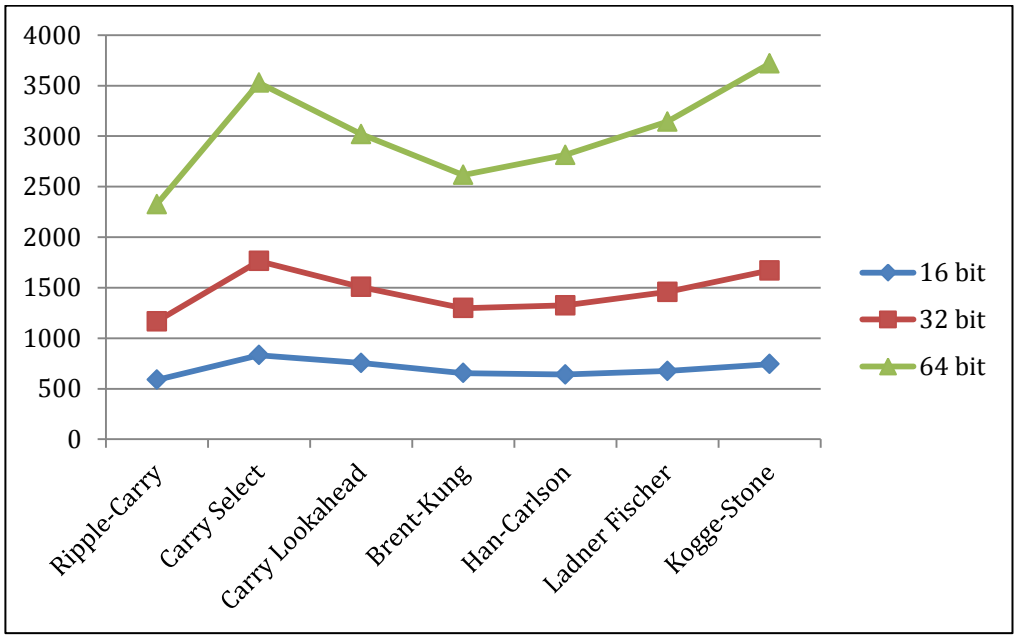

Figure 58: Gate count results

Table 4: Area of Layout

	16 Bit	32 Bit	64 Bit
Ripple Carry	4836.81	8831.85	16383.5
Carry Select	6652.82	13599.24	24243.25
Carry Lookahead	6019.23	11368.39	21448.55
Brent-Kung	4537.33	8575.59	16097.22
Ladner-Fischer	4727.18	9919.49	19650.6
Han-Carlson	4753.16	8928.92	17583.7
Kogge-Stone	5946.37	12326.45	26112.73

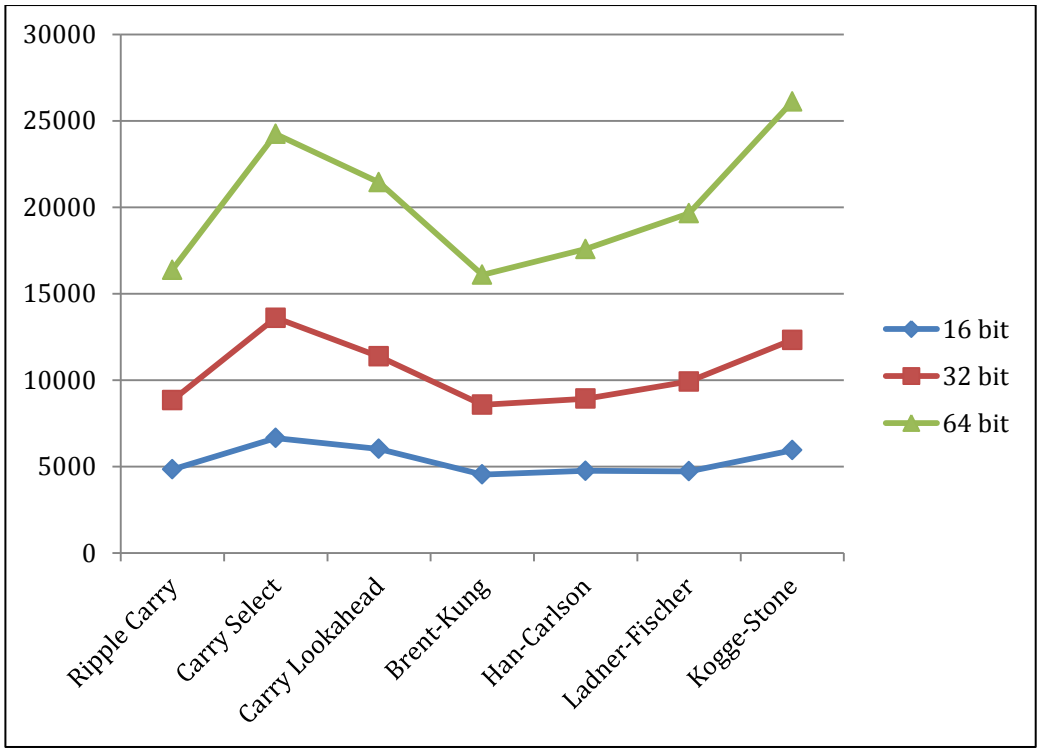

Figure 59: Area of Layout

The table and graph comparing power are given below. Since, all the adders are combinational, the power number doesn't depend on the number of cycles it takes to complete the addition operation and the trend is the same as that of gate count since the total power consumed by the circuit is proportional to the number of gates in the design.

Table 5: Power Results

	16 Bit	**32 Bit**	**64 Bit**
Ripple Carry	2.43E-05	4.87E-05	1.11E-04
Carry Select	3.64E-05	7.12E-05	1.68E-04
Carry Lookahead	3.29E-05	6.33E-05	1.36E-04
Brent-Kung	2.73E-05	5.75E-05	1.21E-04
Han-Carlson	2.37E-05	5.50E-05	1.17E-04
Ladner-Fischer	2.73E-05	6.00E-05	1.28E-04
Kogge-Stone	3.00E-05	6.92E-05	1.56E-04

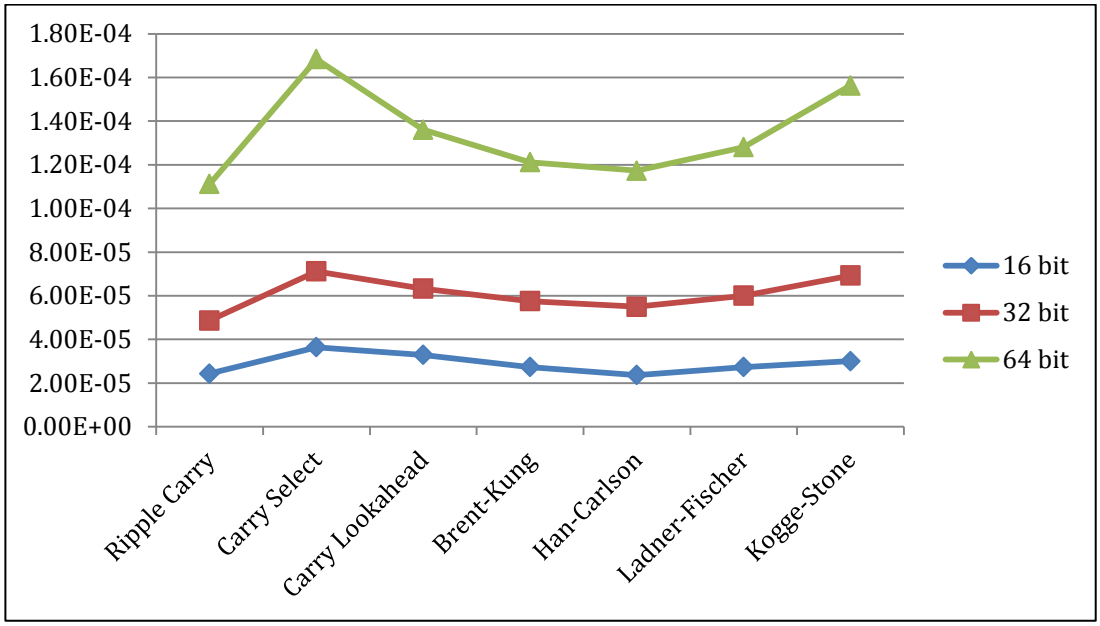

Figure 60: Power Results

The table and chart for Area-Delay product is given below. Ripple carry adders have the worst area-delay product as the delay is very high for them. For carry-select adders, the area is not reduced much and the delay is considerable higher compared to parallel prefix adders and hence their area-delay product is also high. As can be seen from figure 62, Kogge-Stone has the worst Area-Delay product among the parallel prefix adders because of its very high complexity. Next is Brent-Kung because of its worst delay even though it is least complex. Han-Carlson and Ladner-Fischer have very close Area-Delay product for 16 bits but for 32 and 64 bits, Han-Carlson is better than Ladner-Fischer.

Table 6: Area-Delay Product

	16 Bit	**32 Bit**	**64 Bit**
Ripple Carry	11118.38	37181.21	156380.9
Carry Select	11193.37	29976.8	101499.2
Carry Lookahead	9237.11	22424.15	47229.71
Brent-Kung	3399.821	9028.381	21977.53
Ladner-Fischer	3153.502	7977.254	21232.47
Han-Carlson	3156.098	7630.655	18410.13
Kogge-Stone	3907.954	9618.329	25355.46

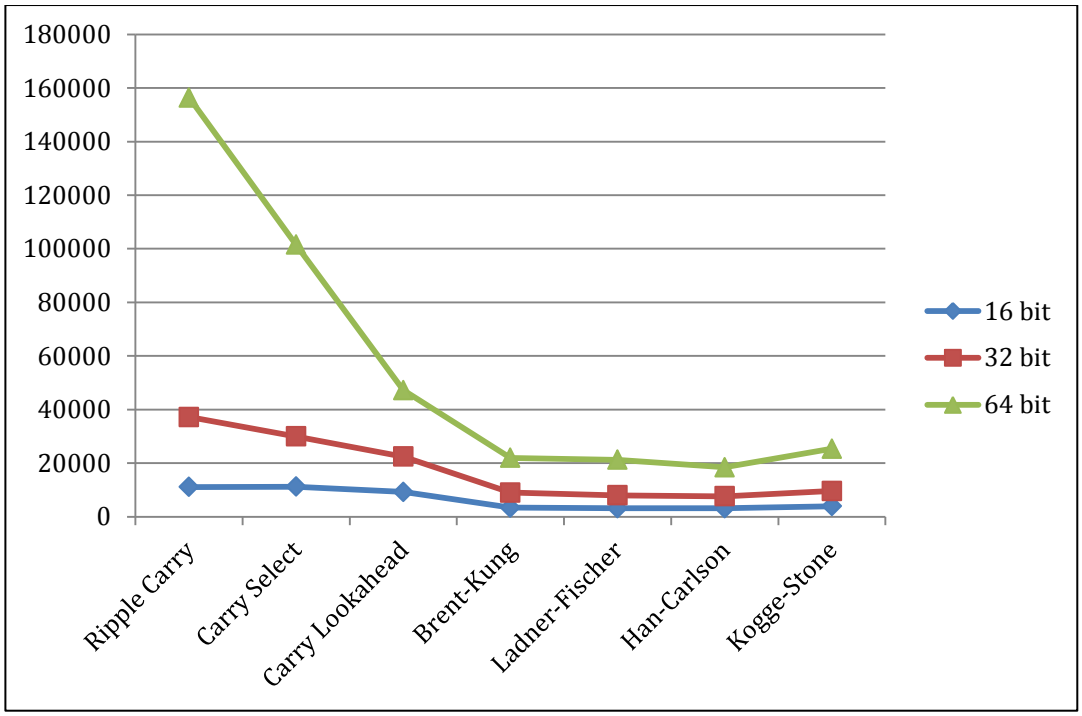

Figure 61: Area-Delay Product Results

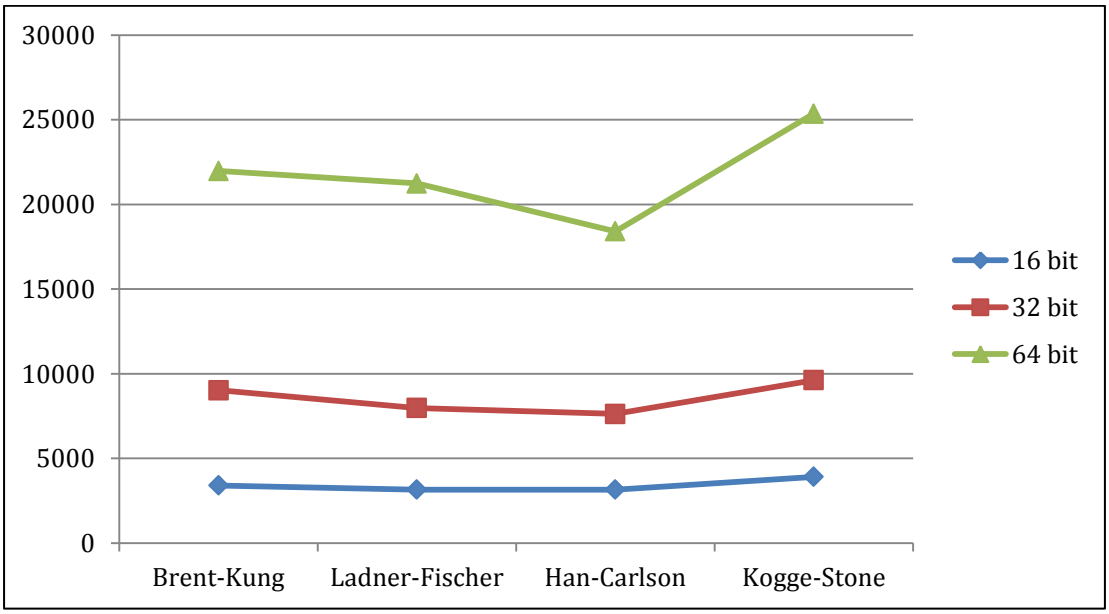

Figure 62: Area-Delay Product for parallel-prefix adders

4. Conclusions

Among all the adders studied, the ripple carry adder was found to have the maximum delay and the lowest gate count which is consistent with expectation as the ripple carry adder is the simplest form of adder. The delay of carry select adder and carry lookahead adder is considerably less compared to ripple carry adders. However, the complexity of carry select adder is quite high since it uses two blocks of ripple carry adders to perform addition and a multiplexer to choose the correct sum and carry. Parallel prefix adders have significantly lower delays as compared to other adders. Among them, the Kogge-Stone adder was found to have the least delay. This came at the cost of occupying the most area for 16 bit, 32 bit and 64 bit versions. This is consistent as Kogge-Stone adder has the most number of parallel prefix operations thereby consuming the maximum area. The fanout at each stage is limited to two and thus the delay for each stage is low. Brent-Kung has the least area since there are very few prefix operations done but has the worst delay among tree adders as it needs more number of stages to do the computation is large compared to Kogge-Stone. Han-Carlson and Ladner-Fischer have delay and area between these two extremes. The delay of Han-Carlson and Ladner-Fischer are approximately same even though Han-Carlson has one more stage as compared to Ladner-Fischer. This is because Lardner-Fischer adder has a large fanout towards the bottom. As a result, the delay proportionately increases even after appropriately buffering the nets. The area of Han-Carlson is slightly less than Ladner-Fisher. Also, Han-Carlson is most efficient in terms of area-delay product.

As a result of this, Kogge-Stone should be used if delay needs to be minimized with no constraint on area. Brent-Kung should be used if area needs to be minimized with some allowable delay. If delay and area both need to be optimized, Han-Carlson should be used. These results also support the claim of Han-Carlson that it is the fastest possible area efficient adder.

5. References

[1] O. J. Bedrij, "Carry-Select Adders", *IRE Transactions on Electronic Computers,* vol. EC-11, pp. 340-346, 1962

[2] A. Weinberger and J. L. Smith, "A logic for high-speed addition," *National Bureau of Standards Circular 591*, pp. 3-12, 1958

[3] O. L. MacSorley, "High Speed Arithmetic in Binary Computers," *Proc. IRE*, vol. 49, pp. 67-91, 1961

[4] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective* (*4th Edition)*, 2011

[5] P. M. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Transactions on Computers*, Vol. C-22, No. 8, pp. 783-791, August 1973

[6] R. E. Lardner and M. J. Fischer, "Parallel Prefix Computation," *Journal of the ACM*, Vol. 27, No. 4, pp. 831-838, October 1980

[7] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Transactions on Computers*, Vol. C-31, No. 3, March 1982

[8] T. Han and D.A. Carlson, "Fast Area-Efficient VLSI Adders," *1987 IEEE 8th Symposium on Computer Arithmeti*c, pp.49-56, May 1987